Créer une application avec FastAPI pour Python

FastAPI est un framework web rapide et léger pour construire des interfaces de programmation d'applications modernes en utilisant Python 3.6 et plus. Dans ce tutoriel, nous allons parcourir les bases de la création d'une application avec FastAPI, et vous comprendrez pourquoi il a été désigné comme l'un des meilleurs frameworks open source de 2021.

Une fois que vous serez prêt à développer vos propres applications FastAPI, vous n'aurez pas à chercher bien loin pour trouver un endroit où les héberger. Les services d'hébergement d'applications (EC2) et d'hébergement de bases de données (RDS) de AWS fournissent une plateforme en tant que service qui est forte en Python.

Commençons par apprendre les bases.

Avantages de FastAPI

Vous trouverez ci-dessous quelques-uns des avantages que le framework FastAPI apporte à un projet.

- **Rapidité :** Comme son nom l'indique, FastAPI est un framework très rapide. Sa vitesse est comparable à celle de Go et Node.js, qui sont généralement considérés comme faisant partie des options les plus rapides pour la création d'API.
- **Facile à apprendre et à coder :** FastAPI a déjà prévu presque tout ce dont vous aurez besoin pour créer une API prête à la production. En tant que développeur utilisant FastAPI, vous n'avez pas besoin de tout coder à partir de zéro. Avec seulement quelques lignes de code, vous pouvez avoir une API RESTful prête à être déployée. **Une documentation complète :** FastAPI
- utilise les normes de documentation OpenAPI, ainsi la documentation peut être générée dynamiquement. Cette documentation fournit des informations détaillées sur les points de terminaison, les réponses, les paramètres et les codes de retour de FastAPI.
- **Des API avec moins de bugs :** FastAPI supporte la validation personnalisée des données, ce qui permet aux développeurs de construire des API avec moins de bugs. Les développeurs de FastAPI se vantent que le framework entraîne moins de bugs d'origine humaine – jusqu'à 40 % de moins.
- **Indices de type :** Le module types a été introduit dans Python 3.5. Il vous permet de déclarer le `type` d'une variable. Lorsque le type d'une variable est déclaré, les IDE sont en mesure de fournir un meilleur support et de prédire les erreurs avec plus de précision.

Avant de démarrer avec FastAPI, nous allons créer un nouveau projet sur Github.

Allez sur votre compte Github et allez dans **repositories (Dépôts).**

Cliquez sur **New** (Nouveau) pour créer un nouveau repository.

Renseignez un nom de repository, exemple : **fastapi-tutorial.**

Cochez la case **Ajouter le fichier README**.

Cliquez sur **create repository**.

Une fois le nouveau repository crée, clonez le repository sur votre ordinateur et ouvrez le dans visual studio code.

Dans Visual studio code, ouvrez un terminal et positionnez vous à l'intérieur du repository **fastapi-tutoriel.**

Comment démarrer avec FastAPI

Pour suivre ce tutoriel et démarrer avec FastAPI, vous devrez d'abord faire quelques petites choses. Assurez-vous que vous avez un éditeur de texte/IDE de programmeur, tel que Visual Studio Code et un outil de gestion de version de code tel que Git.

Il est courant de faire tourner vos applications Python et leurs instances dans des environnements virtuels. Les environnements virtuels permettent d'exécuter simultanément différents ensembles de paquets et configurations, et d'éviter les conflits dus à des versions de paquets incompatibles.

Avant de commencer à utiliser FastAPI nous allons créer un nouveau projet sur GitHub.

Pour créer un environnement virtuel, ouvrez Visual Studio code votre terminal et exécutez cette commande :

```
$ python3 -m venv env
```

Vous devrez également activer l'environnement virtuel. La commande pour cela variera en fonction du système d'exploitation et du shell que vous utilisez. Voici quelques exemples d'activation CLI pour un certain nombre d'environnements :

```
# On Unix or MacOS (bash shell):
/path/to/venv/bin/activate
```

```
# On Unix or MacOS (csh shell):
/path/to/venv/bin/activate.csh

# On Unix or MacOS (fish shell):
/path/to/venv/bin/activate.fish

# On Windows (command prompt):
pathtovenvScriptsactivate.bat

# On Windows
(PowerShell):pathtovenvScriptActivate.ps1
```

(Certains IDE sensibles à Python peuvent également être configurés pour activer l'environnement virtuel actuel.)

Maintenant, installez FastAPI :

```
$ pip3 install fastapi
```

FastAPI est un framework pour construire des API, mais pour tester vos API vous aurez besoin d'un serveur web local. Uvicorn est un serveur web ASGI (Asynchronous Server Gateway Interface) rapide comme l'éclair pour Python qui est idéal pour le développement. Pour installer Uvicorn, exécutez cette commande :

```
$ pip3 install "uvicorn[standard]"
```

Une fois l'installation réussie, créez un fichier nommé **main.py** dans le répertoire de travail de votre projet. Ce fichier sera le point d'entrée de votre application.

Vue d'un projet FastAPI de base dans un IDE.

Un exemple rapide de FastAPI

Vous allez tester votre installation FastAPI en mettant rapidement en place un exemple de point de terminaison. Dans votre fichier **main.py**, collez le code suivant, puis enregistrez le fichier :

```
# main.py
from fastapi import FastAPI app =
FastAPI() @app.get("/")
async def root():
 return {"greeting":"Hello world"}
```

Le code ci-dessus crée un point de terminaison FastAPI basique. Vous trouverez ci-dessous un résumé de ce que fait chaque ligne :

- `from fastapi import FastAPI` : La fonctionnalité de votre API est fournie par la classe Python FastAPI.
- `app = FastAPI()` : Ceci crée une instance FastAPI.
- `@app.get("/")` : Ceci est un décorateur python qui spécifie à FastAPI que la fonction en dessous de lui est en charge de la gestion des requêtes.

- `@app.get("/")` : Ceci est un décorateur qui spécifie la route. Cela crée une méthode `GET` sur la route du site. Le résultat est ensuite renvoyé par la fonction enveloppée. D'autres opérations
- possibles utilisées pour communiquer incluent `@app.post()`, `@app.put()`, `@app.delete()`, `@app.options()`, `@app.head()`, `@app.patch()` et `@app.trace()`.

Dans le répertoire des fichiers, exécutez la commande suivante dans votre terminal pour démarrer le serveur API :

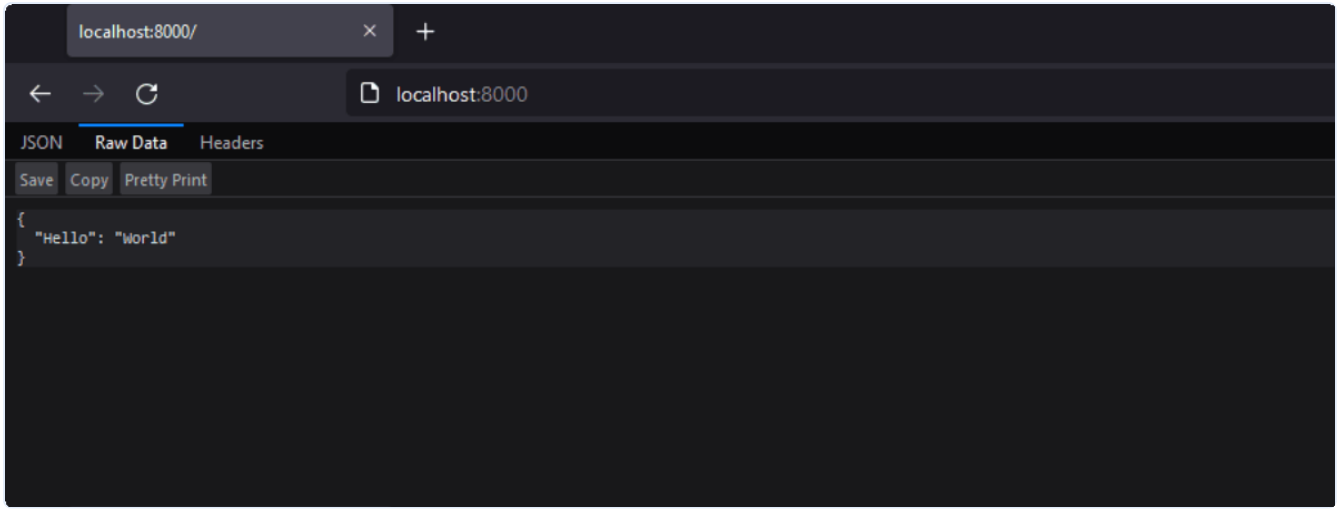
```
$ uvicorn main:app --reload
```

Dans cette commande, `main` est le nom de votre module. L'objet `app` est une instance de votre application, et est importé dans le serveur ASGI. L'indicateur `--reload` indique au serveur de se recharger automatiquement lorsque vous apportez des modifications.

Vous devriez voir quelque chose comme ceci dans votre terminal :

```
 $ uvicorn main:app --reload
INFO: Will watch for changes in these directories: ['D:\WEB
DEV\Eunit\Test INFO: Uvicorn running on http://127.0.0.1:8000
(Press CTRL+C to quit)  INFO: Started reloader process [26888]
using WatchFiles
INFO: Started server process [14956]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Dans votre navigateur, naviguez vers `http://localhost:8000` pour confirmer que votre API fonctionne. Vous devriez voir « Hello » : « World » comme un objet JSON sur la page. Ceci illustre à quel point il est facile de créer une API avec FastAPI. Tout ce que vous aviez à faire était de définir une route et de retourner votre dictionnaire Python, comme vu à la ligne six du code ci-dessus.



Application FastAPI Hello World dans un navigateur.

Utilisation des indices de type

Si vous utilisez Python, vous avez l'habitude d'annoter les variables avec des types de données de base tels que `int`, `str`, `float` et `bool`. Toutefois, à partir de la version 3.9 de Python, des structures de données avancées ont été introduites. Cela vous permet de travailler avec des structures de données telles que `dictionaries`, `tuples`, et `lists`. Avec les indications de type de FastAPI, vous pouvez structurer le schéma de vos données en utilisant des modèles pydantiques et ensuite, utiliser les modèles pydantiques pour faire des indications de type et bénéficier de la validation des données qui est fournie.

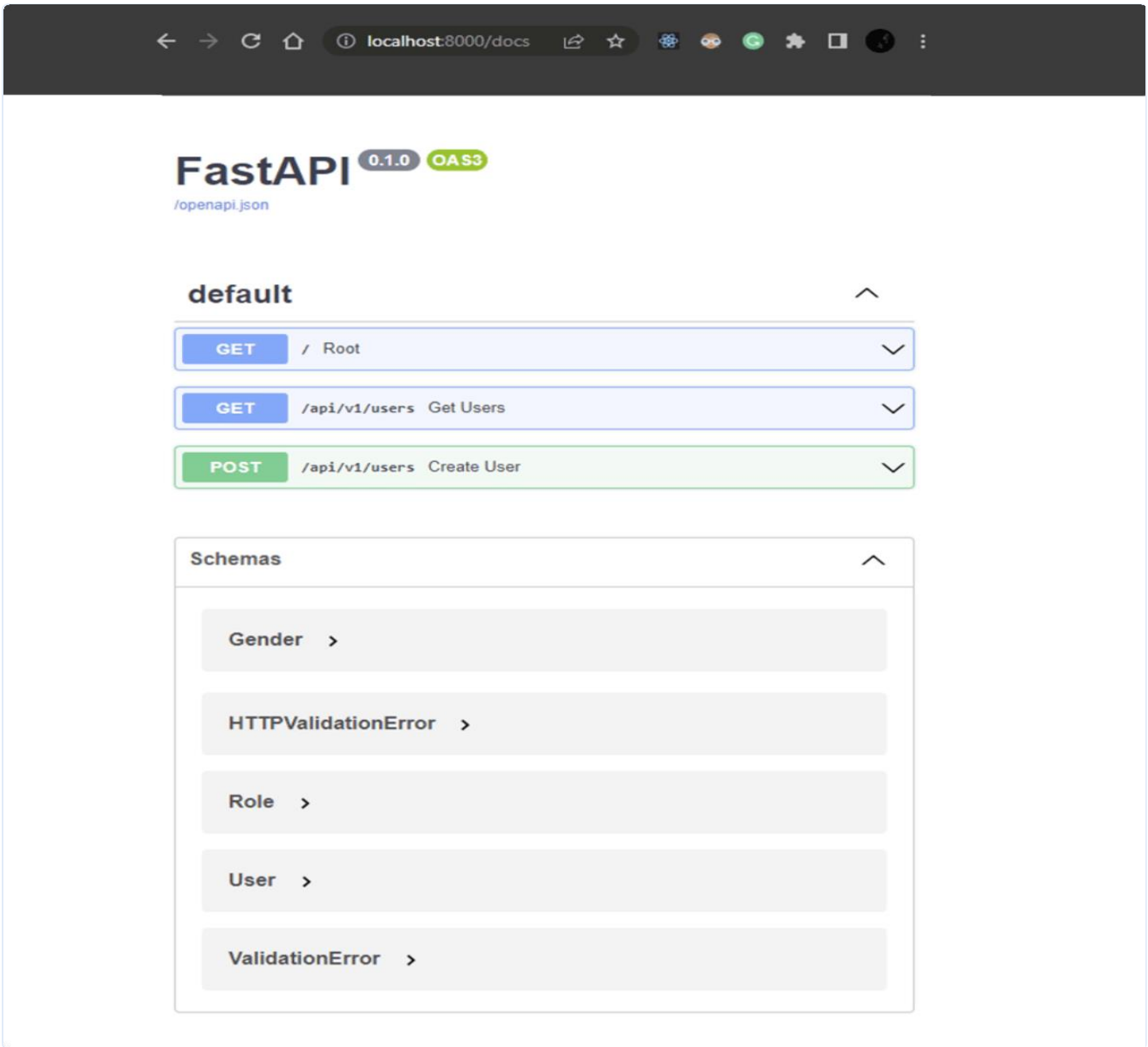
Dans l'exemple ci-dessous, l'utilisation des indications de type en Python est démontrée avec une simple calculatrice de prix de repas, `calculate_meal_fee` :

Notez que les indications de type ne changent pas la façon dont votre code s'exécute.

```
def calculate_meal_fee(beef_price: int, meal_price: int) -> int:
 total_price: int = beef_price + meal_price
 return total_price
print("Calculated meal fee", calculate_meal_fee(75, 19)
```

Documentation interactive de l’API FastAPI

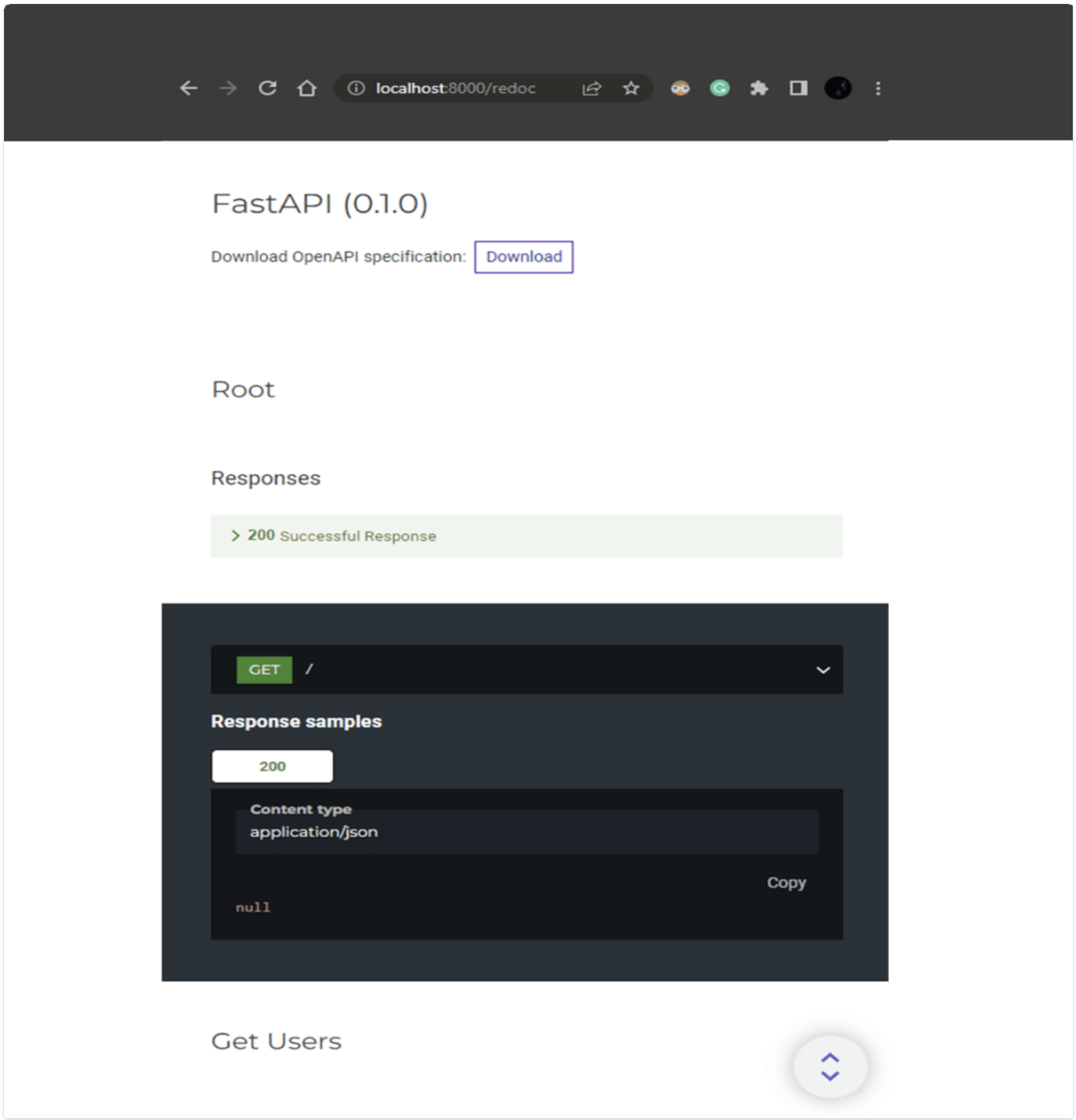
FastAPI utilise Swagger UI pour fournir une documentation API interactive automatique. Pour y accéder, naviguez vers `http://localhost:8000/docs` et vous verrez un écran avec tous vos points de terminaison, méthodes et schémas.



La documentation de Swagger UI pour FastAPI

Cette documentation automatique de l'API basée sur le navigateur est fournie par FastAPI, et vous n'avez rien d'autre à faire pour en profiter.

Une autre documentation API basée sur le navigateur, également fournie par FastAPI, est Redoc. Pour accéder à Redoc, naviguez sur `http://localhost:8000/redoc`, où vous sera présentée une liste de vos points de terminaison, des méthodes et de leurs réponses respectives.



La documentation de Redoc UI pour FastAPI

Configuration des routes dans FastAPI

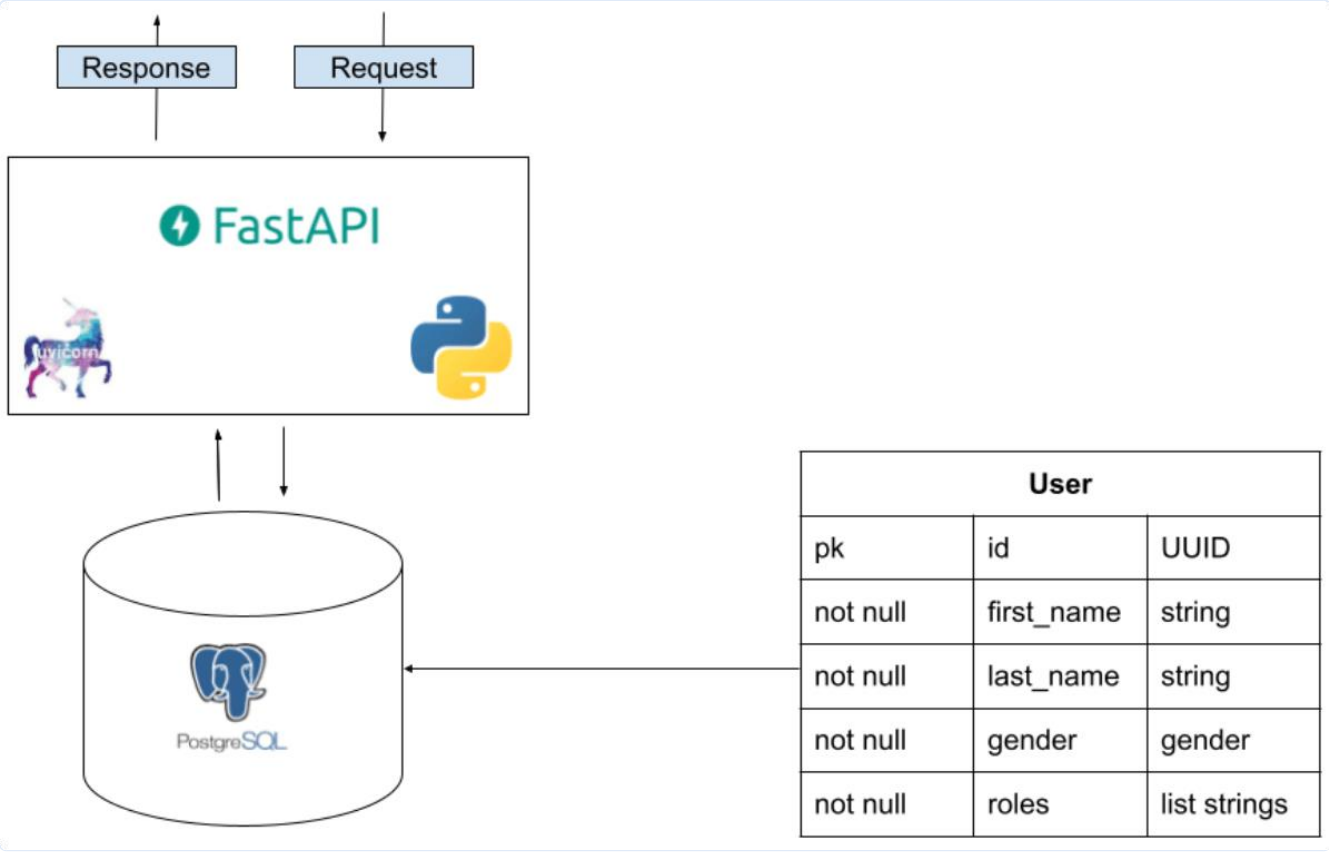
Le décorateur `@app` vous permet de spécifier la méthode deroute, telle que `@app.get` ou `@app.post`, et supporte `GET`, `POST`, `PUT` et `DELETE`, ainsi que les options moins courantes, `HEAD`, `PATCH` et `TRACE`.

Construire votre application avec FastAPI

Dans ce tutoriel, vous serez guidé dans la construction d'une application CRUD avec FastAPI. L'application sera capable de :

- Créer un utilisateur
- Lire l'enregistrement d'un utilisateur dans la base de données
- Mettre à jour un utilisateur existant
- Supprimer un utilisateur particulier

Pour exécuter ces opérations CRUD, vous créerez des méthodes qui exposent les points de terminaison API. Le résultat sera une base de données en mémoire qui peut stocker une liste d'utilisateurs.



Vous utiliserez la bibliothèque pydantic pour effectuer la validation des données et la gestion des réglages en utilisant les annotations de type Python. Pour les besoins de ce tutoriel, vous déclarerez la forme de vos données comme des classes avec des attributs.

Ce tutoriel utilisera la base de données en mémoire. Ceci afin de vous permettre de commencer rapidement à utiliser FastAPI pour construire vos API. Cependant, pour la production, vous pouvez utiliser n'importe quelle base de données de votre choix, comme PostgreSQL, MySQL, SQLite, ou même Oracle.

Construire l'application

Vous commencerez par créer votre modèle d'utilisateur. Le modèle utilisateur aura les attributs suivants :

- `id` : Un identifiant unique universel (UUID)
- `first_name` : Le prénom de l'utilisateur
- `last_name` : Le nom de famille de l'utilisateur
- `gender` : Le sexe de l'utilisateur
- `roles`qui est une liste contenant les rôles `admin` et `user`

Commencez par créer un nouveau fichier nommé **models.py** dans votre répertoire de travail, puis collez le code suivant dans **models.py** pour créer votre modèle :

```
# models.py
from typing import List, Optional
from uuid import UUID, uuid4
from pydantic import BaseModel
from enum import Enum
from pydantic import BaseModel
class Gender(str, Enum):
 male = "male"
 female = "female"
class Role(str, Enum):
 admin = "admin"
 user = "user"
class User(BaseModel):
 id: Optional[UUID] = uuid4()
 first_name: str
 last_name:    str
 gender:    Gender
 roles: List[Role]
```

Dans le code ci-dessus :

- Votre classe `User` étend `BaseModel`, qui est ensuite importée de `pydantic`.
- Vous avez défini les attributs de l'utilisateur, comme indiqué ci-dessus.

L'étape suivante consiste à créer votre base de données. Remplacez le contenu de votre fichier **main.py** par le code suivant :

```
# main.py
from typing import List
from uuid import uuid4
from fastapi import FastAPI
from models import Gender, Role, User
app = FastAPI()
db: List[User] = [
 User(
 id=uuid4(),
 first_name="John",
 last_name="Doe",
 gender=Gender.male,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="Jane",
 last_name="Doe",
 gender=Gender.female,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="James",
 last_name="Gabriel",
 gender=Gender.male,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="Eunit",
 last_name="Eunit",
 gender=Gender.male,
 roles=[Role.admin, Role.user],
 ),
]
```

Dans **main.py** :

- Vous avez initialisé `db` avec un type de `List`, et transmis le modèle `User`
- Vous avez créé une base de données en mémoire avec quatre utilisateurs, chacun ayant les attributs requis tels que `first_name`, `last_name`, `gender` et `roles`. L'utilisateur `Eunit` se voit attribuer les rôles de `admin` et `user`, tandis que les trois autres utilisateurs se voient attribuer uniquement le rôle de `user`.

Lire les enregistrements de la base de données

Vous avez réussi à configurer votre base de données en mémoire et à la remplir d'utilisateurs. L'étape suivante consiste donc à configurer un point de terminaison qui renverra une liste de tous les utilisateurs. C'est là que FastAPI entre en jeu.

Dans votre fichier **main.py**, collez le code suivant juste en dessous de votre point de terminaison `Hello World` :

```
# main.py
 @app.get("/api/v1/users")
 async def get_users():
 return db
```

Ce code définit le point de terminaison `/api/v1/users`, et crée une fonction asynchrone, `get_users`, qui renvoie tout le contenu de la base de données, `db`.
Sauvegardez votre fichier, et vous pouvez tester votre point de terminaison utilisateur.
Exécutez la commande suivante dans votre terminal pour démarrer le serveur API :

```
$ uvicorn main:app --reload
```

Dans votre navigateur, naviguez vers `http://localhost:8000/api/v1/users`. Cela devrait renvoyer une liste de tous vos utilisateurs, comme indiqué ci-dessous

Données d'utilisateur récupérées par une requête de lecture de la base de données FastAPI.

À ce stade, votre fichier **main.py** ressemblera à ceci :

```
# main.py
from typing import List
from uuid import uuid4
from fastapi import FastAPI
from models import Gender, Role, User
app = FastAPI()
db: List[User] = [
 User(
 id=uuid4(),
 first_name="John",
 last_name="Doe",
 gender=Gender.male,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="Jane",
 last_name="Doe",
 gender=Gender.female,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
 first_name="James",
 last_name="Gabriel",
 gender=Gender.male,
 roles=[Role.user],
 ),
 User(
 id=uuid4(),
```

```
 first_name="Eunit",
 last_name="Eunit",
 gender=Gender.male,
 roles=[Role.admin, Role.user],
 ),
]
@app.get("/")
async def root():
 return {"Hello": "World",}
@app.get("/api/v1/users")
async def get_users():
 return db
```

Créer des enregistrements de base de données

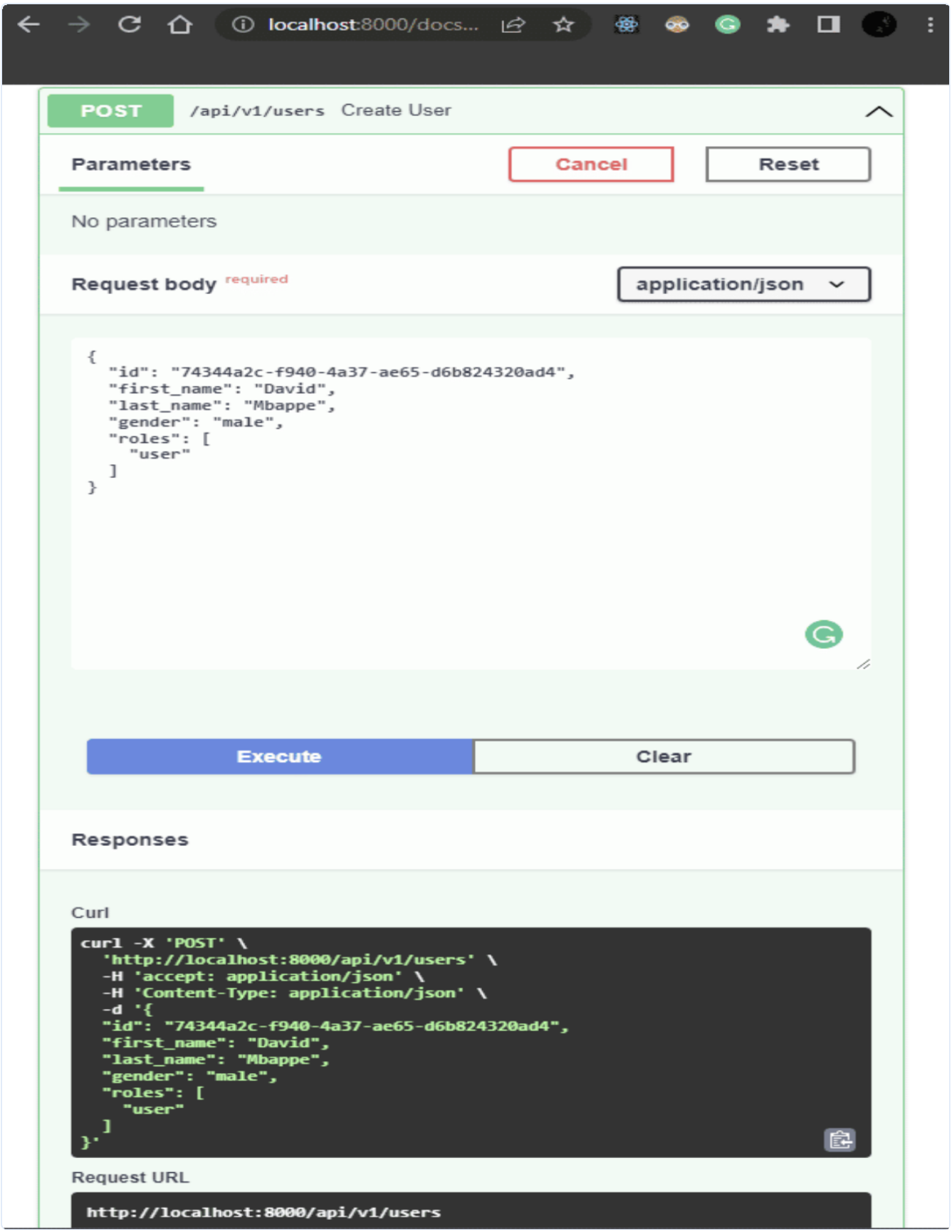
L'étape suivante consiste à créer un point de terminaison pour créer un nouvel utilisateur dans votre base de données. Collez le code suivant dans votre fichier **main.py** :

```
# main.py
@app.post("/api/v1/users")
async def create_user(user: User):
 db.append(user)
 return {"id": user.id}
```

Dans ce code, vous avez défini le point de terminaison pour soumettre un nouvel utilisateur et utilisé le décorateur `@app.post` pour créer une méthode `POST`.

Vous avez également créé la fonction `create_user`, qui accepte `user` du modèle `User`, et ajouté (appended) le nouveau `user` créé à la base de données, `db`. Enfin, le point de terminaison renvoie un objet JSON de l'utilisateur nouvellement créé `id`.

Vous devrez utiliser la documentation API automatique fournie par FastAPI pour tester votre point de terminaison, comme vu ci-dessus. Ceci est dû au fait que vous ne pouvez pas faire une requête POST en utilisant le navigateur web. Naviguez vers `http://localhost:8000/docs` pour tester en utilisant la documentation fournie par SwaggerUI.


localhost:8000/docs...
POST
/api/v1/users
Create User
Parameters
Cancel
Reset
No parameters
Request body
required
application/json
{
"id": "74344a2c-f940-4a37-ae65-d6b824320ad4",
"first_name": "David",
"last_name": "Mbappe",
"gender": "male",
"roles": [
"user"
]
}
Execute
Clear
Responses
Curl
curl -X 'POST' \
'http://localhost:8000/api/v1/users' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"id": "74344a2c-f940-4a37-ae65-d6b824320ad4",
"first_name": "David",
"last_name": "Mbappe",
"gender": "male",
"roles": [
"user"
]
}'
Request URL
http://localhost:8000/api/v1/users

— Paramètres pour une requête POST FastAPI.

Supprimer des enregistrements de base de données

Puisque vous construisez une application CRUD, votre application devra avoir la possibilité de supprimer une ressource spécifiée. Pour ce tutoriel, vous allez créer un point de terminaison pour supprimer un utilisateur.

Collez le code suivant dans votre fichier **main.py** :

```
# main.py
from uuid import UUID
from fastapi HTTPException
@app.delete("/api/v1/users/{id}")
async def delete_user(id: UUID):
for user in db:
 if user.id == id:
 db.remove(user)
 return
raise HTTPException(
 status_code=404, detail=f"Delete user failed, id {id} not found."
 )
```

Voici une décomposition ligne par ligne du fonctionnement de ce code :

- `@app.delete("/api/v1/users/{id}")` : Vous avez créé le point de terminaison de suppression à l'aide du décorateur `@app.delete()`. Le chemin est toujours `/api/v1/users/{id}`, mais il récupère ensuite `id`, qui est une variable de chemin correspondant à l'id de l'utilisateur.
- `async def delete_user(id: UUID):` : Crée la fonction `delete_user`, qui récupère le `id` à partir de l'URL.

- `for user in db:` : Ceci indique à l'application de boucler à travers les utilisateurs dans la base de données, et de vérifier si le `id` passé correspond à un utilisateur dans la base de données.
- `db.remove(user)` : Si le `id` correspond à un utilisateur, l'utilisateur sera supprimé ; sinon, un `HTTPException` avec un code d'état de 404 sera levé.


localhost:8000/doc...
DELETE
/api/v1/users/{id}
Delete User
Parameters
Cancel
Name
Description
id * required
string($uuid)
(path)
18d3f517-5008-46fb-9519-e2d5669b1dd3
Execute
Clear
Responses
Curl
curl -X 'DELETE' \
'http://localhost:8000/api/v1/users/18d3f517-5008-46fb-9519-e2d5669b1dd3'
-H 'accept: application/json'
Request URL
http://localhost:8000/api/v1/users/18d3f517-5008-46fb-9519-e2d5669b1dd3
Server response
Code
Details
200
Response body
null
Download
Response headers
content-length: 4
content-type: application/json
date: Tue,30 Aug 2022 02:50:20 GMT
server: uvicorn
Responses
Code
Description
Links
200
Successful Response
No links

— Paramètres pour une requête FastAPI DELETE.

Mise à jour des enregistrements de la base de données

Vous allez créer un point de terminaison pour mettre à jour les détails d'un utilisateur. Les détails qui peuvent être mis à jour incluent les paramètres suivants : `first_name`, `last_name`, et `roles`.

Dans votre fichier **models.py**, collez le code suivant sous votre modèle `User`, c'est-à-dire après la classe `User(BaseModel):` :

```
# models.py
class UpdateUser(BaseModel):
first_name: Optional[str]
last_name: Optional[str]
roles: Optional[List[Role]]
```

Dans cet extrait, la classe `UpdateUser` étend `BaseModel`. Vous définissez ensuite les paramètres utilisateur pouvant être mis à jour, tels que `first_name`, `last_name`, et `roles`, comme étant facultatifs.

Vous allez maintenant créer un point de terminaison pour mettre à jour les détails d'un utilisateur particulier. Dans votre fichier **main.py**, collez le code suivant après le décorateur `@app.delete` :

```
# main.py
@app.put("/api/v1/users/{id}")
async def update_user(user_update: UpdateUser, id: UUID):
```

```
    for user in db:
    if user.id == id:
    if user_update.first_name is not None:
    user.first_name = user_update.first_name
    if user_update.last_name is not None:
    user.last_name = user_update.last_name
    if user_update.roles is not None:
    user.roles = user_update.roles
    return user.id
    raise HTTPException(status_code=404, detail=f"Could not find user with
    id
```

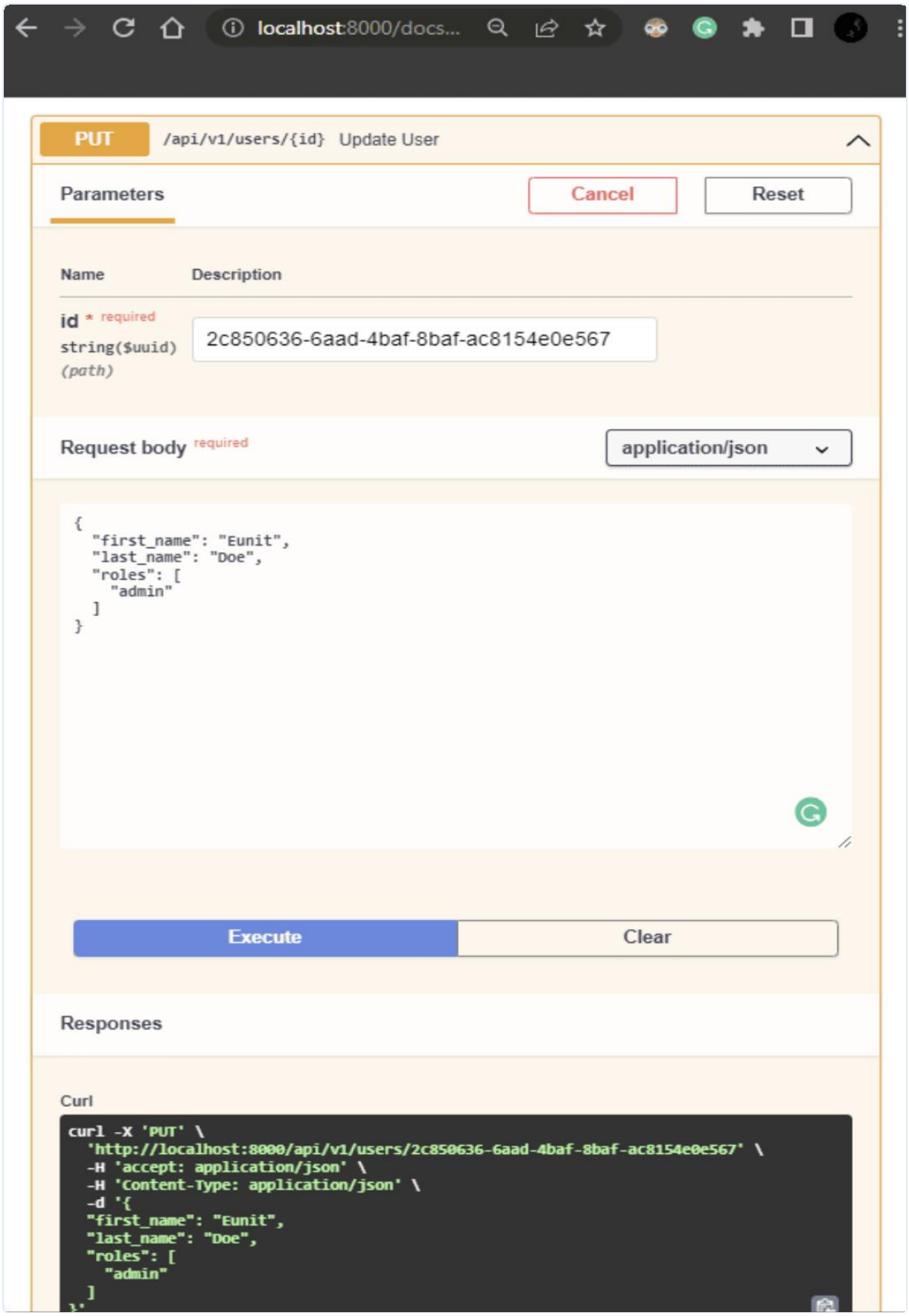
Dans le code ci-dessus, vous avez fait ce qui suit :

- Création de `@app.put("/api/v1/users/{id}")`, le point de terminaison de mise à jour. Il a un paramètre variable `id` qui correspond à l'id de l'utilisateur.
- Création d'une méthode appelée `update_user`, qui prend en compte la classe `UpdateUser` et `id`.
- Utilisation d'une boucle `for` pour vérifier si l'utilisateur associé à la passe `id` est dans la base de données.
- Vérification si l'un des paramètres de l'utilisateur est `is not None` (non nul). Si un paramètre, tel que `first_name`, `last_name`, ou `roles`, n'est pas nul, il est mis à jour.
- Si l'opération réussit, l'identifiant de l'utilisateur est renvoyé.
- Si l'utilisateur n'a pas été localisé, une exception `HTTPException` avec un code d'état de 404 et un message de `Could not find user with id: {id}` est levée.

Pour tester ce point de terminaison, assurez-vous que votre serveur Uvicorn est en cours d'exécution. S'il n'est pas en cours d'exécution, saisissez cette commande :

```
uvicorn main:app --reload
```

Vous trouverez ci-dessous une capture d'écran du test.


localhost:8000/docs...
PUT
/api/v1/users/{id} Update User
Parameters
Cancel
Reset
Name
Description
id * required
string($uuid)
(path)
2c850636-6aad-4baf-8baf-ac8154e0e567
Request body required
application/json
{
"first_name": "Eunit",
"last_name": "Doe",
"roles": [
"admin"
]
}
Execute
Clear
Responses
Curl
curl -X 'PUT' \
'http://localhost:8000/api/v1/users/2c850636-6aad-4baf-8baf-ac8154e0e567' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"first_name": "Eunit",
"last_name": "Doe",
"roles": [
"admin"
]
}'

— Paramètres pour une requête UPDATE FastAPI.

Résumé

Dans ce tutoriel, vous vous êtes familiarisé avec le framework FastAPI pour Python et avez constaté par vous-même la rapidité avec laquelle vous pouvez mettre en place une application alimentée par FastAPI. Vous avez appris à construire des points de terminaison API CRUD à l’aide du framework – création, lecture, mise à jour et suppression d’enregistrements de base de données.