FastAPI y MongoDB

Crea una Aplicación CRUD con FastAPI y MongoDB

En este tutorial, usted aprendera como desarrollar una API asincrónica con FastAPI and MongoDB. Nosotros estaremos usando el paquete Motor para interactuar sincrónicamente con MongoDB.

Objectivos

Al final de este tutorial, podrás:

1. Desarrollar una RESTful API con Python and FastAPI.
2. Interactuar con MongoDB sincrónicamente.
3. Ejecutar MongoDB en la cloud con MongoDB Atlas
4. Implementar una aplicación FastAPI en Heroku

Configuración Inicial

Comience creando una nueva carpeta para guardar su proyecto llamado "fastapi-mongo":

```
mkdir fastapi-mongo
cd fastapi-mongo
```

A continuación, crea y activa el ambiente virtual:

```
python3.9 -m venv venv
source venv/bin/activate
export PYTHONPATH=$PWD
```

> Sientete libre de cambiar virtualenv y Pip por Poetry o Pipenv. Por mas, revisa Modern Python Environments.

A continuación, crea los siguientes archivos y folders:

```
├── apps
|   ├── __init__.py
|   ├── main.py
|   └── server
|       ├── app.py
|       ├── database.py
|       ├── models
|       └── routes
└── requirements.txt
```

Agregue las siguientes dependencias a su archivo *requirements.txt*:

```
fastapi
uvicorn
```

Instalarlos

```
pip install -r requirements.txt
```

El el archivo app/main.py, define un punto de entrada para ejecutar :

```
import uvicorn

if __name__ == "__main__":
    uvicorn.run("apps.server.app:app", host="0.0.0.0", port=8000, reload=True)
```

Aqui, le indicamos al archivo que ejecute un servidor Uvicorn en el puerto 8000 y se vuelva a cargar en cada cambio de archivo.
Antes de iniciar el servidor atraves del punto de entrada, crea un ruta(route) base en el archivo *app/server/app.py*:

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/", tags=["Root"])
```

```
async def read_root():
    return {"message": "Welcome to this fantastic app!"}
```

Tags son identificadores para agrupar rutas(routes). Las rutas con las mismas etiquetas se agrupan en una seccion de la documentación de la API.

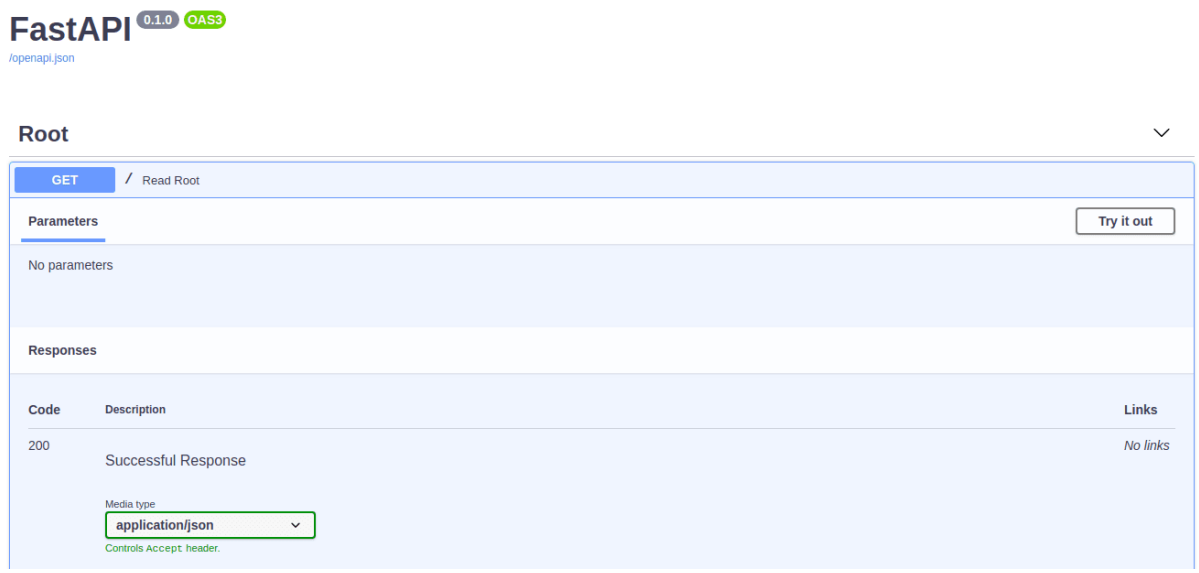
Ejecute el archivo de punto de entrada desde su consola:

```
python apps/main.py
```

Navega a http://localhost:8000 en su navegador. Deberías ver:

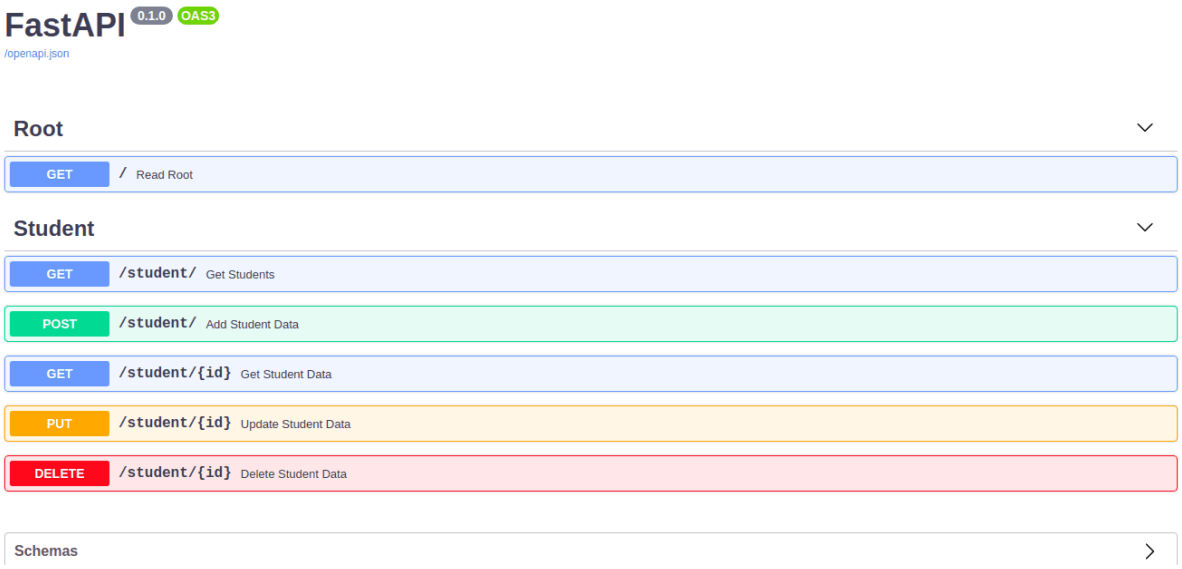
```
{
  "message": "Welcome to this fantastic app!"
}
```

podrás ver la documentación de la API interactiva en http://localhost:8000/docs:



Rutas (Routes)

Nosotros construiremos una aplicación simple para almacenar datos de estudiantes con las siguientes rutas(routes) CRUD:



Antes de sumergirnos en la escritura de las rutas (routes), primero definamos el esquema relevante y configuremos MongoDB.

Esquema (Schema)

Definamos el Esquema (Schema) en el que se basarán nuestros datos, que representará cómo se almacenan los datos en la base de datos de MongoDB.

Los esquemas de Pydantic se utilizan para validar datos junto con serializar (JSON -> Python) y deserializar (Python -> JSON). En otras palabras, no sirve como schema validator de mongo.

El la carpeta "app/server/models", crea un archivo llamado *student.py*:

```
import code
from email import message
from msilib import schema
from typing import Optional
from pydantic import  BaseModel, EmailStr, Field

class StudentSchema(BaseModel):
    fullname: str = Field(...)
    email: EmailStr = Field(...)
    course_of_study = str = Field(...)
    year: int = Field(..., gt=0, lt=9)
    gpa: float = Field(..., le=4.0)
```

```
    class Config:
        schema_extra = {
            "example":{
                "fullname": "John Doe",
                "email": "jdoe@qccc.edu",
                "course_of_study": "Water resources engineering",
                "year": 2,
                "gpa": "3.0",
            }
        }

class UpdateStudentModel(BaseModel):
    fullname: Optional[str]
    email: Optional[str]
    course_of_study: Optional[str]
    year: Optional[str]
    gpa: Optional[str]

    class config:
        schema_extra = {
            "example": {
                "fullname": "John Doe",
                "email": "jdoe@qccc.edu",
                "course_of_study": "Water resources and enviromental engeneering",
                "year": 2,
                "gpa": "3.0",
            }
        }

class UpdateStudentModel(BaseModel):
    fullname: Optional[str]
    email: Optional[str]
    course_of_study: Optional[str]
    year: Optional[str]
    gpa: Optional[str]

    class Confid:
        schema_extra = {
            "example": {
                "fullname": "John Doe",
                "email": "jdoe@.qccc.edu",
                "course_of_study": "Water resources and environmental engineering",
                "year": 4,
                "gpa": "4.0",
            }
        }
```

```
def ResponseModel(data, message):
    return {
        "data": [data],
        "code": 200,
        "message": message,
    }

def ErrorResponseModel(error, code, message):
    return {
        "error": error,
        "code": code,
        "message": message,
    }
```

En el código de arriba, nosotros vamos a definir una Pydantic Schema llamado StudentSchema que representa cómo se almacenarán los datos de los estudiantes en su base de datos MongoDB
En Pydantic, los puntos suspensivos ellipsis, ..., esto indica que este campo es requerido. este tambien puede ser remplazado con None o un valor predeterminado. En el StudentSchema, cada campo tiene puntos suspensivos, ya que cada campo es importante y el programa no debe continuar sin tener los valores establecidos.

En el campo gpa y year en la StudentSchema, nosotros agregamos los validatorsgt, lt, y le:

1. gt y lt en el campo del year aseguran que el valor que pasado sea mayor que 0 y menor que 9. Como resultado, valores como 0, 10, 11 generarán errores.
2. El validador en el campo gpa se asegura que el valor que pasado sea igual o menor que 4.0.

Este esquema ayudará a los usuarios a enviar solicitudes HTTP con la forma adecuada a la API, es decir, el tipo de datos a enviar y cómo enviarlos.

FastAPI usa Pyantic Schemas para documentar automáticamente modelos de datos junto con Json Schema. Luego la interfaz de usuario Swagger UI representa los datos

de los modelos de datos generados. Puede leer más sobre cómo FastAPI genera documentación API aqui.
Como usamos EmailStr, necesitamos instalar el validador de correo email-validator.

Agregue las siguientes dependencias a su archivo *requirements.txt*:

```
pydantic[email]
```

Instalarlos:

```
pip install -r requirements.txt
```

Con el esquema en su lugar, configuremos MongoDB antes de escribir las rutas para la API.

MongoDB

En esta sección, conectaremos MongoDB y configuraremos nuestra aplicación para comunicarse con élla.

Segun Wikipedia, MongoDB es un programa de base de datos orientado a documentos multiplataforma. Clasificado como un programa de base de datos NoSQL, MongoDB utiliza documentos similares a JSON con esquemas opcionales.

MongoDB Configuration

Si no tiene MongoDB instalado en su máquina, consulte la guía de Installation de los documentos. Una vez instalado, continúe con la guía para ejecutar el proceso mongod daemon daemon. Una vez hecho esto, puede verificar que MongoDB esté funcionando conectándose a la instancia a través del comando mongo shell:

```
$ mongo
```

Como referencia, este tutorial utiliza MongoDB Community Edition v5.0.6.

```
$ mongo --version

MongoDB shell version v5.0.6

Build Info: {
    "version": "5.0.6",
    "gitVersion": "212a8dbb47f07427dae194a9c75baec1d81d9259",
    "modules": [],
    "allocator": "system",
    "environment": {
        "distarch": "x86_64",
        "target_arch": "x86_64"
    }
}
```

Motor Configuration

A continuación, configuraremos Motor, un controlador MongoDB asíncrono, para interactuar con la base de datos.

Comience agregando la dependencia al archivo de requisitos requirements.txt:

```
motor
```

Instalarlos:

```
pip install -r requirements.txt
```

De vuelta en la aplicación, agregue la información de conexión de la base de datos al archivo *app/server/database.py*:

```
import motor.motor_asyncio

MONGO_DETAILS = "mongodb://localhost:27017"

client = motor.motor_asyncio.AsyncIOMotorClient(MONGO_DETAILS)
```

```
database = client.students

student_collection = database.get_collection("students_collection")
```

En el código anterior, importamos Motor, definimos los detalles de la conexión y creamos un cliente a través de AsyncIOMotorClient.

Luego hicimos referencia a una base de datos llamada students y una colección (similar a una tabla en una base de datos relacional) llamada students_collection. Dado que estas son solo referencias y no I/O(reales, tampoco requiere un await expression. When the first I/O se realiza la operación, tanto la base de datos como la colección se crearán si aún no existen.

A continuación, cree una función de ayuda rápida para analizar los resultados de una consulta de base de datos en un dictado de Python.

Agregue esto también al archivo app/server/database.py:

```
import motor.motor_asyncio

MONGO_DETAILS = "mongodb://localhost:27017"

client = motor.motor_asyncio.AsyncIOMotorClient(MONGO_DETAILS)

database = client.students

student_collection = database.get_collection("students_collection")

#helper student function
def student_helper(student)-> dict:
    return {
        "id": str(student["_id"]),
        "fullname": student["fullname"],
        "email": student["email"],
        "course_of_study": student["course_of_study"],
        "year": student["year"],
        "GPA": student["gpa"],
    }
```

A continuación, escribamos las operaciones de la base de datos CRUD.

Operaciones CRUD de base de datos

Comience importando el método ObjectId del paquete bson en la parte superior del archivo app/server/*database.py* file:

```
from bson.objectid import ObjectId
```

bson viene instalada como una dependencia del motor comes installed as a dependency of motor.

A continuación, agregue cada una de las siguientes funciones para las operaciones CRUD:

```
#helper student function
def student_helper(student)-> dict:
    return {
        "id": str(student["_id"]),
        "fullname": student["fullname"],
        "email": student["email"],
        "course_of_study": student["course_of_study"],
        "year": student["year"],
        "GPA": student["gpa"],
    }

# Recuperar todos los estudiantes presentes en la base de datos
async def retrieve_students():
    students = []
    async for student in student_collection.find():
        students.append(student_helper(student))
    return students

# Agregar un nuevo estudiante a la base de datos
async def add_student(student_data: dict) -> dict:
    student = await student_collection.insert_one(student_data)
    new_student = await student_collection.find_one({'_id': student.insert_id})
    return student_helper(new_student)

# Recuperar un estudiante con una identificación coincidente
async def retrieve_student(id: str) -> dict:
    student = await student_collection.find_one({'_id': ObjectId(id)})
    if student:
```

```
        return student_helper(student)

# Actualizar a un estudiante con un ID coincidente
async def update_student(id: str, data: dict) -> dict:
    # Retorna falso si se envía un cuerpo de solicitud vacío.
    if len(data) < 1:
        return False
    student = await student_collection.find_one({'_id': ObjectId(id)})
    if student:
        update_student = await student_collection.update_one({'_id': ObjectId(id)},
{'$set': data})
        if update_student:
            return True
        return False

# Eliminar un estudiante de la base de datos
async def delete_student(id: str):
    student = await student_collection.find_one({'_id': ObjectId(id)})
    if student:
        await student_collection.delete_one({'_id': ObjectId(id)})
        return False
```

En el código anterior, definimos las operaciones asincrónicas para crear, leer, actualizar y eliminar datos de estudiantes en la base de datos a través de motor.

En las operaciones de actualización y borrado se busca al alumno en la base de datos para decidir si se realiza o no la operación. Los valores devueltos guían cómo enviar respuestas al usuario, en lo que trabajaremos en la siguiente sección.

CRUD Routes (Rutas)

En esta sección, agregaremos las rutas para complementar las operaciones de la base de datos en el archivo de la base de datos.

En la carpeta "rutas", cree un nuevo archivo llamado student.py y agréguele el siguiente contenido:

```
from fastapi import APIRouter, Body
from fastapi.encoders import jsonable_encoder

from apps.server.database import (
```

```
    add_student,
    delete_student,
    retrieve_student,
    retrieve_students,
    update_student,
)
from apps.server.models.student import (
    ErrorResponseModel,
    ResponseModel,
    StudentSchema,
    UpdateStudentModel,
)

router = APIRouter()
```

Usaremos el JSON Compatible Encoder de FastAPI para convertir nuestros modelos a un formato que sea compatible con JSON.

Luego, conecta la ruta del estudiante en app/server/app.py:

```
from fastapi import FastAPI

from apps.server.routes.student import router as StudentRouter

app = FastAPI()

app.include_router(StudentRouter, tags=["Student"], prefix="/student")

@app.get("/", tags=["Root"])
async def read_root():
    return {"message": "Welcome to this fantastic app!"}
```

Create (Crear)

De vuelta en el archivo de rutas, agregue el siguiente controlador para crear un nuevo student.py:

```
@router.post("/", response_description="Student data added into the database")
async def add_student_data(student: StudentSchema = Body(...)):
    student = jsonable_encoder(student)
```

```
    new_student = await add_student(student)
    return ResponseModel(new_student, "Student added successfully.")
```

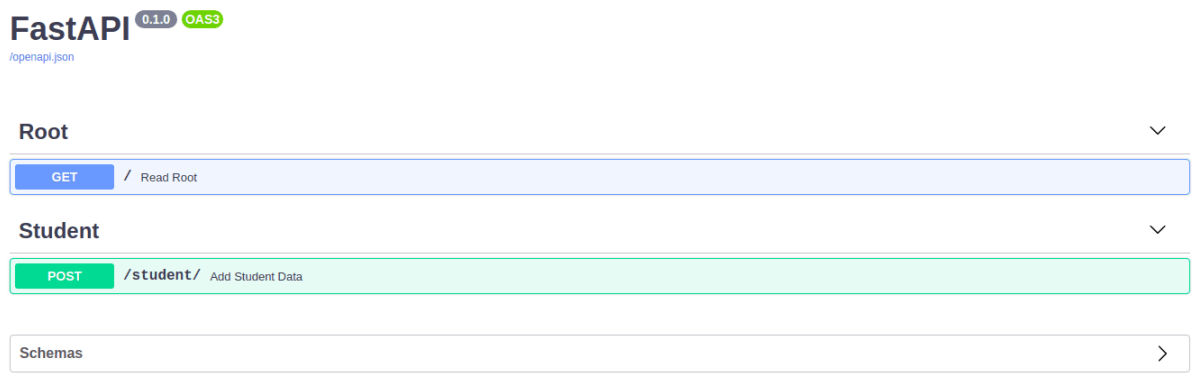
Entonces, la ruta espera una carga útil que coincida con el formato de StudentSchema. Ejemplo:

```
{
    "fullname": "John Doe",
    "email": "jdoe@x.edu.ng",
    "course_of_study": "Water resources engineering",
    "year": 2,
    "gpa": "3.0",
}
```

Inicie el servidor Uvicorn:

```
python apps/main.py
```

Y actualice la página de documentación de la API interactiva en http://localhost:8000/docs para ver la nueva ruta:

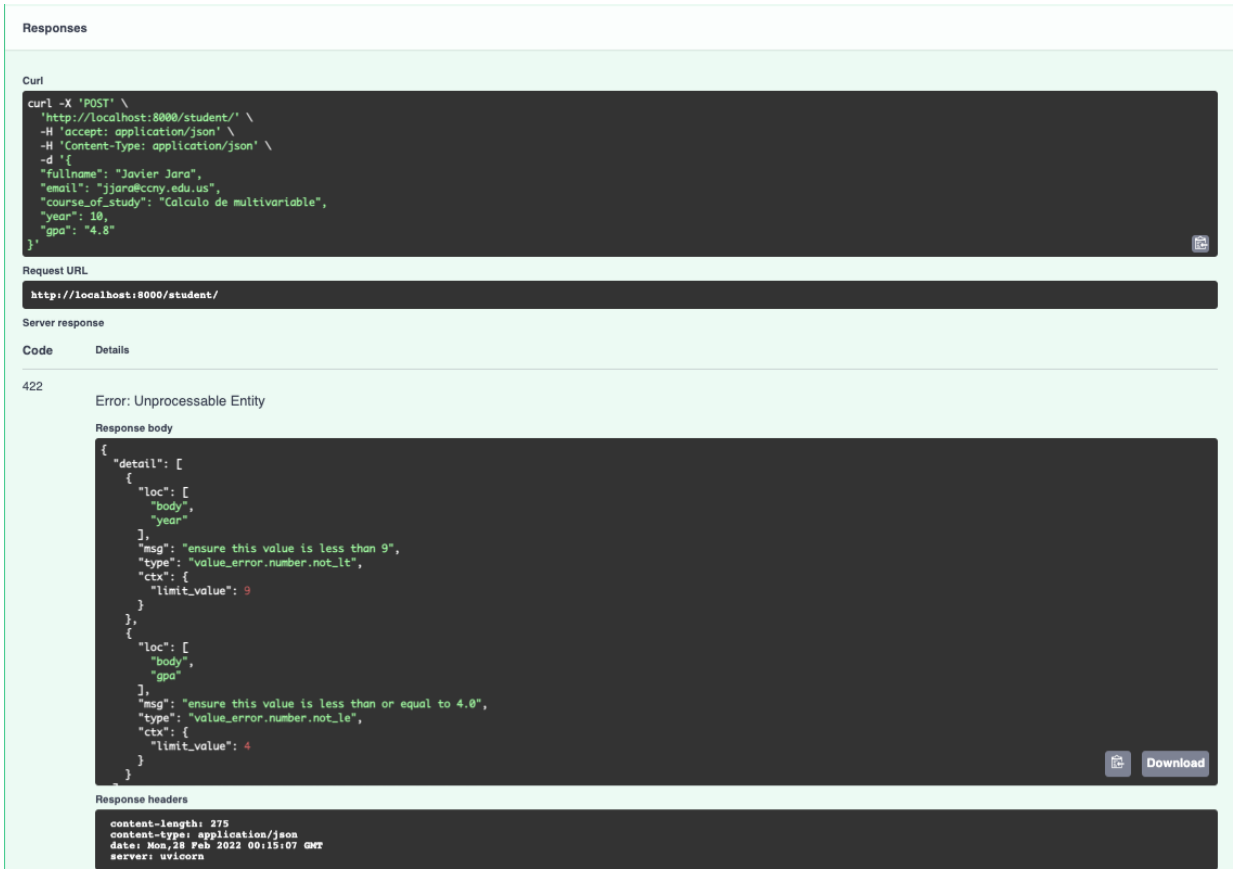


Pruébalo también:

Por lo tanto, cuando se envía una solicitud al punto final, almacena un cuerpo de solicitud codificado en JSON en la variable student antes de llamar antes de llamar al método de base de datos
add_student y almacenar la respuesta en la variable new_student. La respuesta de la base de datos se devuelve a través de ResponseModel.

Pruebe también los validadores:

1. Year debe ser mayor que 0 y menor que 10
2. GPA debe ser menor o igual a 4.0

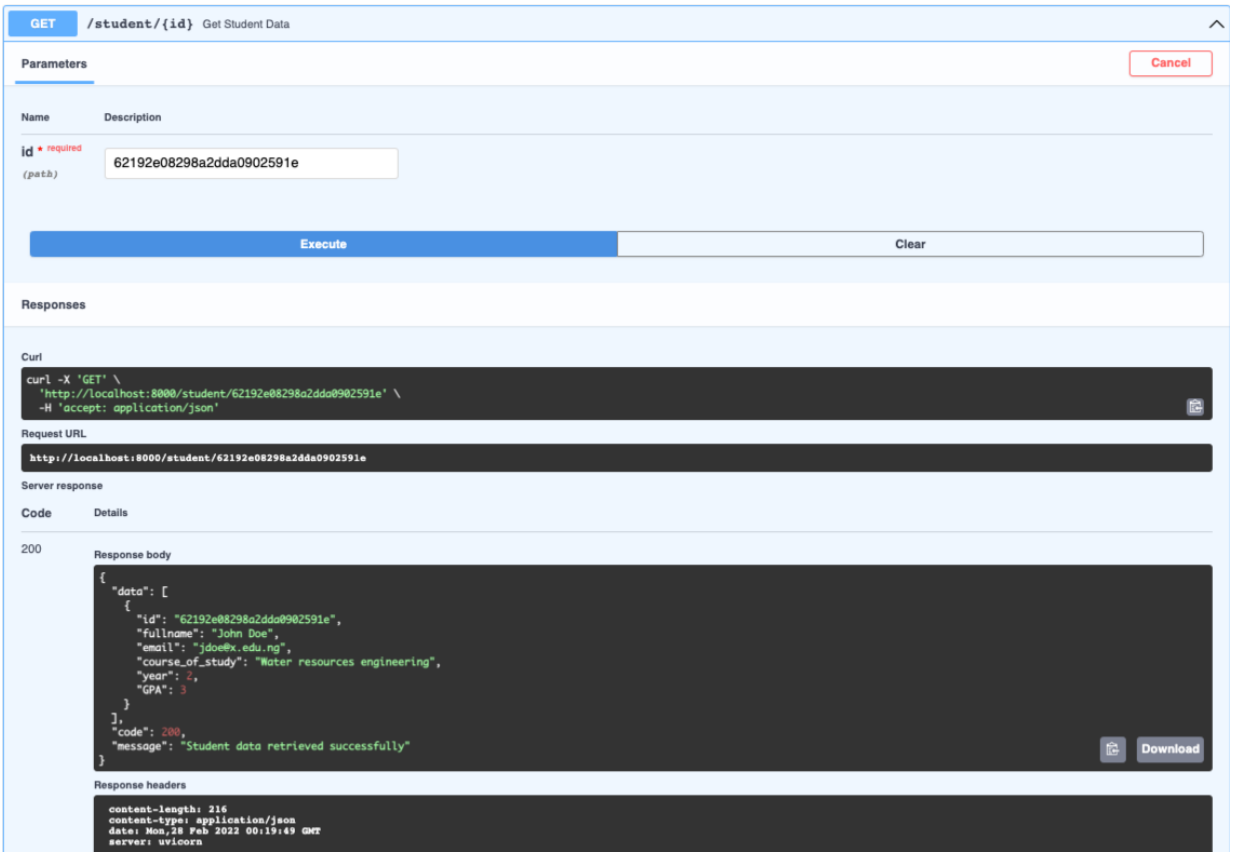


Read (Leer)

Siguiendo adelante, agregue las siguientes rutas para recuperar a todos los estudiantes y a un solo estudiante:

```
@router.get("/", response_description="Students retrieved")
async def get_students():
    students = await retrieve_students()
    if students:
        return ResponseModel(students, "Students data retrieved successfully")
    return ResponseModel(students, "Empty list returned")

@router.get("/{id}", response_description="Student data retrieved")
async def get_student_data(id):
    student = await retrieve_student(id)
    if student:
        return ResponseModel(student, "Student data retrieved successfully")
    return ErrorResponseModel("An error occurred.", 404, "Student doesn't exist.")
```



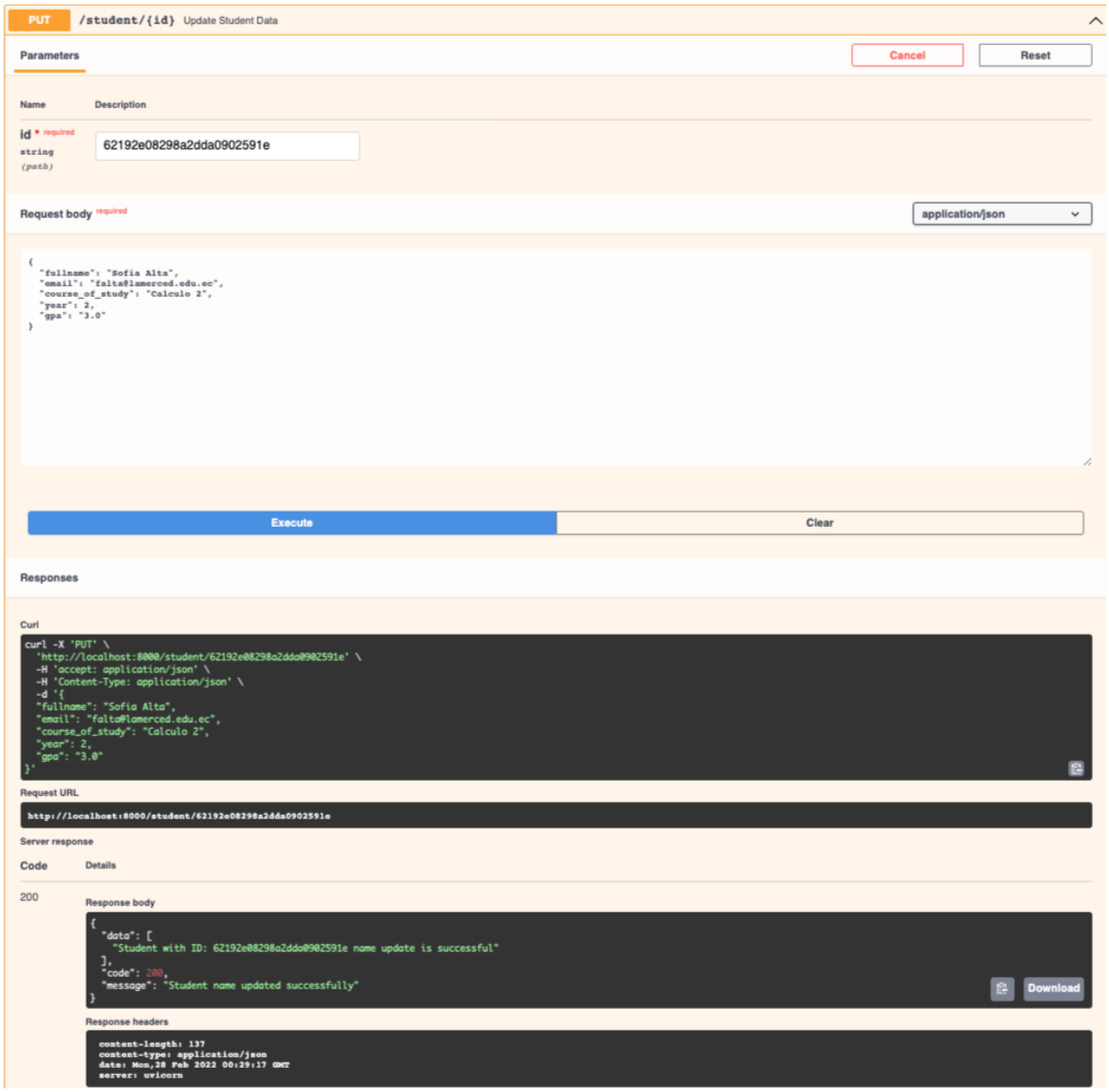
Qué sucede si no pasa un ObjectId válido, por ejemplo, 1, para que el ID recupere la ruta de un solo estudiante? ¿Cómo puedes manejar mejor esto en la aplicación?. Cuando se implemente la operación de eliminación, tendrá la oportunidad de probar la respuesta para una base de datos vacía.

Update (Actualizar)

A continuación, escriba la ruta individual para actualizar los datos del estudiante:

```
@router.put("/{id}")
async def update_student_data(id: str, req: UpdateStudentModel = Body(...)):
    req = {k: v for k, v in req.dict().items() if v is not None}
    updated_student = await update_student(id, req)
    if updated_student:
        return ResponseModel(
            "Student with ID: {} name update is successful".format(id),
            "Student name updated successfully",
        )
```

```
    return ErrorResponseModel(
        "An error occurred",
        404,
        "There was an error updating the student data.",
    )
```

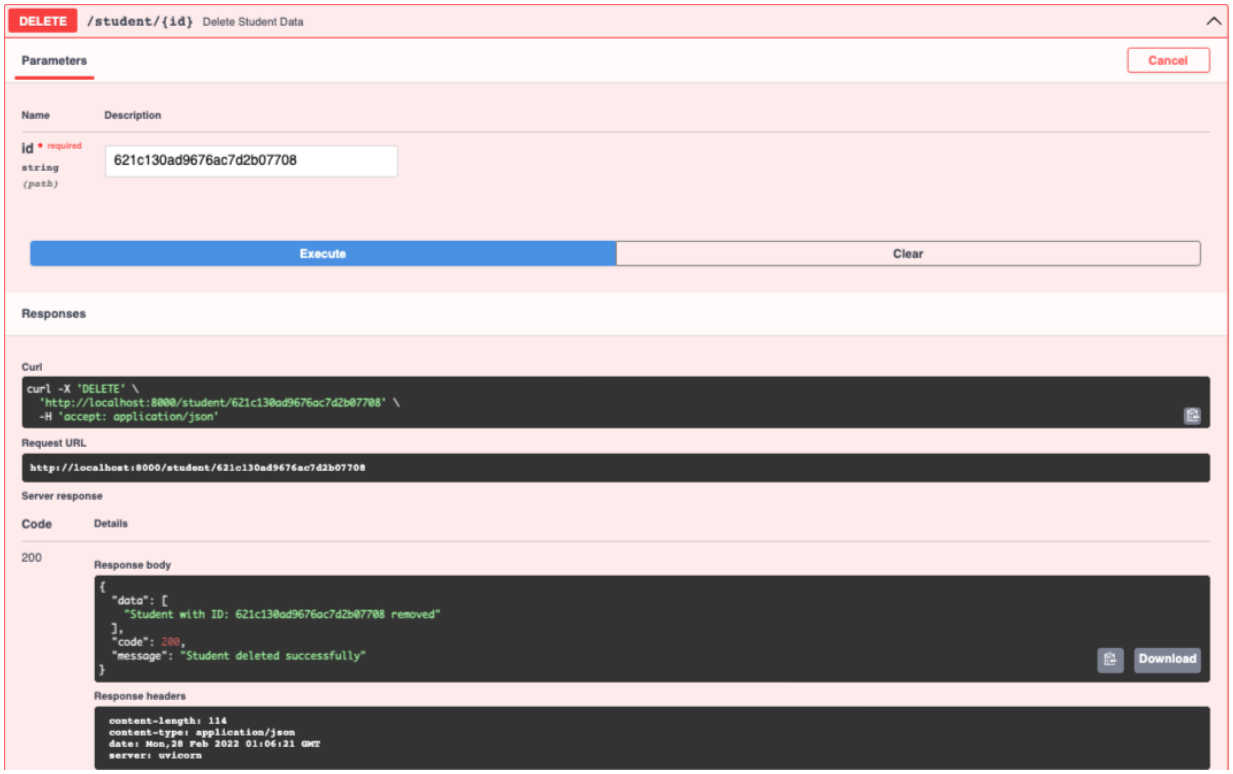


Delete (Borrar)

Finalmente, agregue la ruta de eliminación:

```
@router.delete("/{id}", response_description="Student data deleted from the
database")
async def delete_student_data(id: str):
    deleted_student = await delete_student(id)
    if deleted_student:
        return ResponseModel(
            "Student with ID: {} removed".format(id), "Student deleted successfully"
        )
    return ErrorResponseModel(
        "An error occurred", 404, "Student with id {0} doesn't exist".format(id)
    )
```

Recupere el ID del estudiante que creaste anteriormente y pruebe la ruta de eliminación:



Elimine a los estudiantes restantes y pruebe las rutas de lectura nuevamente, asegurándose de que las respuestas sean apropiadas para una base de datos vacía.

Deployment (Implementación)

En esta sección, implementaremos la aplicación en Heroku y configuraremos una base de datos en la nube para MongoDB.

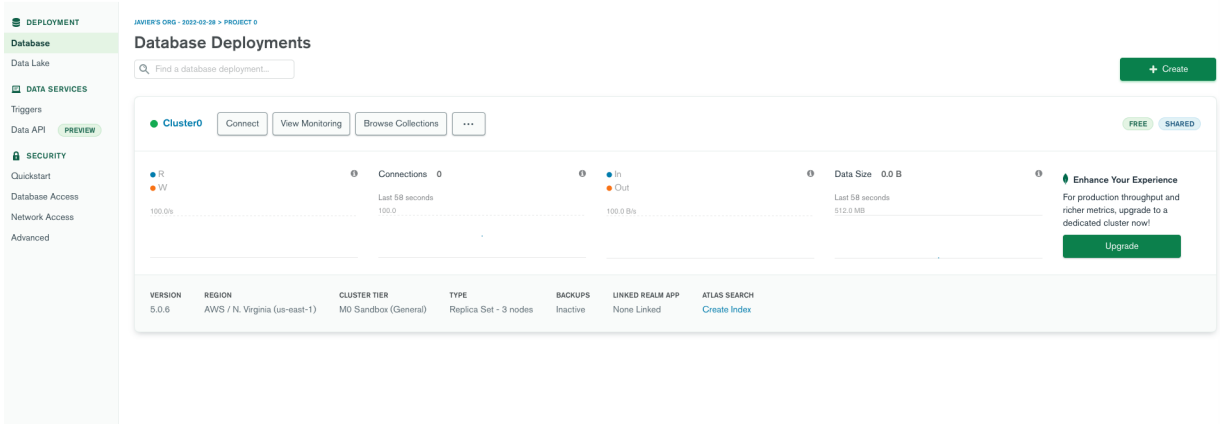
MongoDB Atlas

Antes de implementar, debemos configurar MongoDB Atlas, un servicio de base de datos en la nube para que MongoDB aloje nuestra base de datos..

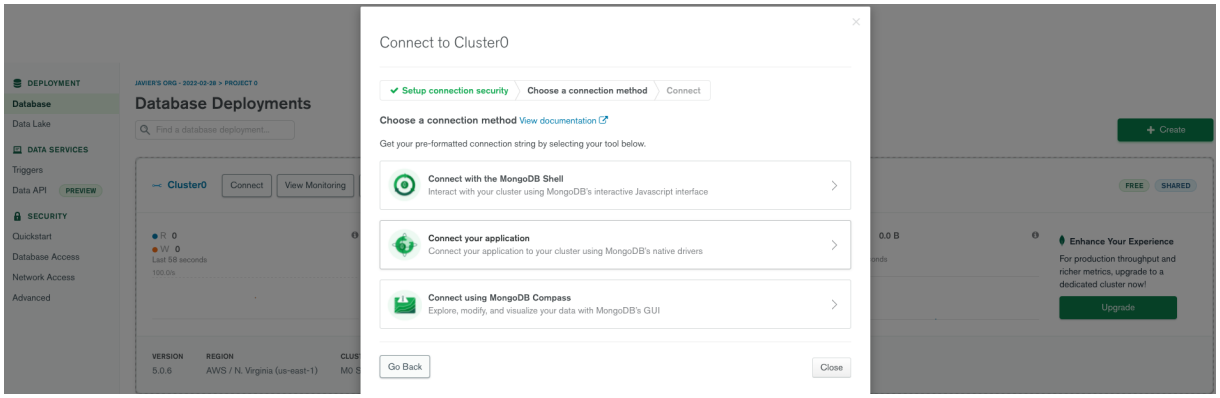
Siga la Guía de Introducción donde creará una cuenta, implementará un clúster de nivel gratuito, configurará un usuario y agregará una dirección IP a la lista blanca.

Para fines de prueba, use 0.0.0.0/0 para la IP incluida en la lista blanca para permitir el acceso desde cualquier lugar. Para una aplicación de producción, querrá restringir el acceso a una IP estática.

Una vez hecho esto, obtenga la información de conexión de la base de datos de su clúster haciendo clic en el botón "Connect":



Haz clic en la segunda opción, "Connect to your application":



Copy the connection URL, making sure to update the password. Set the default database to "students" as well. It will look similar to:

```
mongodb+srv://dbUser:<your_password>@cluster0.cyud5.mongodb.net/students?
retryWrites=true&w=majority
```

En lugar de codificar este valor en nuestra aplicación, definiremos que tiene una variable de entorno. Cree un nuevo archivo llamado .env en la raíz del proyecto y la información de conexión:

```
MONGO_DETAILS=your_connection_url
```

Asegúrese de reemplazar your_connection_url con la URL copiada.

A continuación, para simplificar la gestión de las variables de entorno en nuestra aplicación, instalemos el paquete Python Decouple. Agréguelo a su archivo de requisitos así:

```
python-decouple
```

Instalarlos:

```
pip install -r requirements.txt
```

En el archivo app/server/database.py, importe la biblioteca:

```
from decouple import config
```

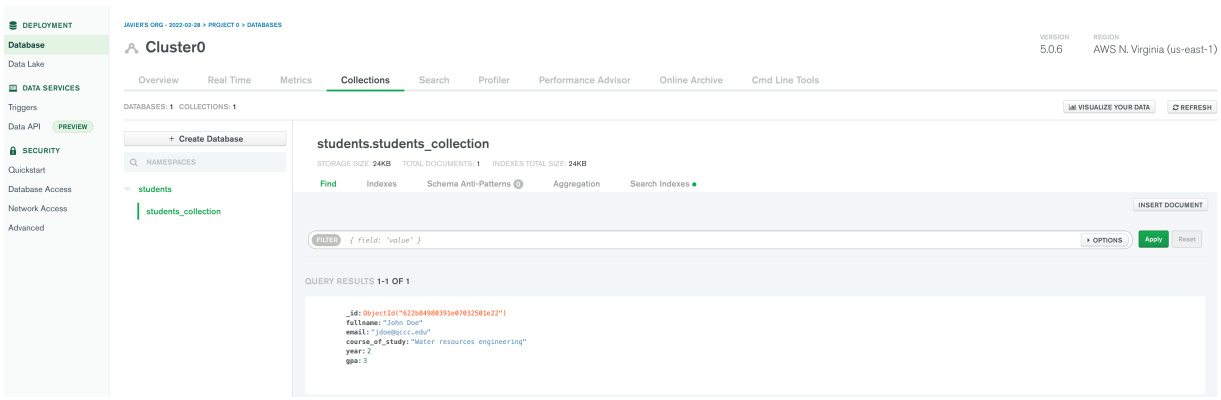
El método de config importado busca en el directorio raíz un archivo .env y lee el contenido que se le pasa. Entonces, en nuestro caso, leerá la variable MONGO_DETAILS.
A continuación, cambie la variable MONGO_DETAILS a:

```
MONGO_DETAILS = config("MONGO_DETAILS") # lee lq variable del environment (.env)
```

Pruebas locales (Testing Locally)

Antes de implementar, probemos la aplicación localmente con la base de datos en la nube para asegurarnos de que la conexión esté configurada correctamente. Reinicie su servidor Uvicorn y pruebe cada ruta desde la documentación interactiva en http://localhost:8000/docs.

Debería poder ver los datos en el tablero de Atlas:



Implementación en Heroku (Deploying to Heroku)

Finalmente, implementemos la aplicación en Heroku.

Heroku es una plataforma en la nube como servicio (PaaS) que se utiliza para implementar y escalar aplicaciones.

Si es necesario, registrese para obtener una cuenta Heroku e instale el Heroku CLI.

Antes de continuar, cree un archivo .gitignore en el proyecto para evitar registrar la carpeta "venv" y el archivo .env en git:

```
touch .gitignore
```

Add the following:

```
# Environments
.env
venv/

# Byte-compiled / optimized / DLL files
__pycache__

# Pyre type checker
.pyre/
.DS_Store
```

A continuación, agregue un Procfile a la raíz de su proyecto:

```
web: uvicorn apps.server.app:app --host 0.0.0.0 --port=$PORT
```

Notas:

1. El Procfile es un archivo de texto, ubicado en la raíz de su proyecto, que guía a Heroku sobre cómo ejecutar su aplicación. Como estamos sirviendo una aplicación web, definimos el tipo de proceso de web junto con el comando para servir Uvicorn.
2. Heroku expone dinámicamente un puerto para que su aplicación se ejecute en el momento de la implementación, que se expone a través de la variable de entorno $PORT.

Su proyecto ahora debería tener los siguientes archivos y carpetas:

```
├── .env
├── .gitignore
├── Procfile
├── LICENSE
├── apps
|   ├── __init__.py
|   ├── main.py
|   └── server
|       ├── app.py
|       ├── database.py
|       ├── models
|       |   └── student.py
|       └── routes
|           └── student.py
├── README.md
└── requirements.txt
```

En la raíz de tu proyecto, inicializa un nuevo repositorio git:
Asumo que git ya esta instalado en la Computadora y pueden publicar al repositorio publico de git

```
git init
git add .
git commit -m "Mi primera fastapi and mongo applicacion"
```

Ahora, podemos crear una nueva aplicación en Heroku:

```
heroku login
heroku create
```

Junto con la creación de una nueva aplicación, este comando crea un repositorio git remoto en Heroku para que podamos impulsar nuestra aplicación para su implementación. Luego establece esto como un control remoto en el repositorio local automáticamente para nosotros.
Puede verificar que el control remoto esté configurado ejecutandogit remote -v

Tome nota de la URL de su aplicación.

Como no agregamos el archivo .env a git, debemos configurar la variable de entorno dentro del entorno de Heroku:

```
heroku config:set MONGO_DETAILS="your_connection_url"
```

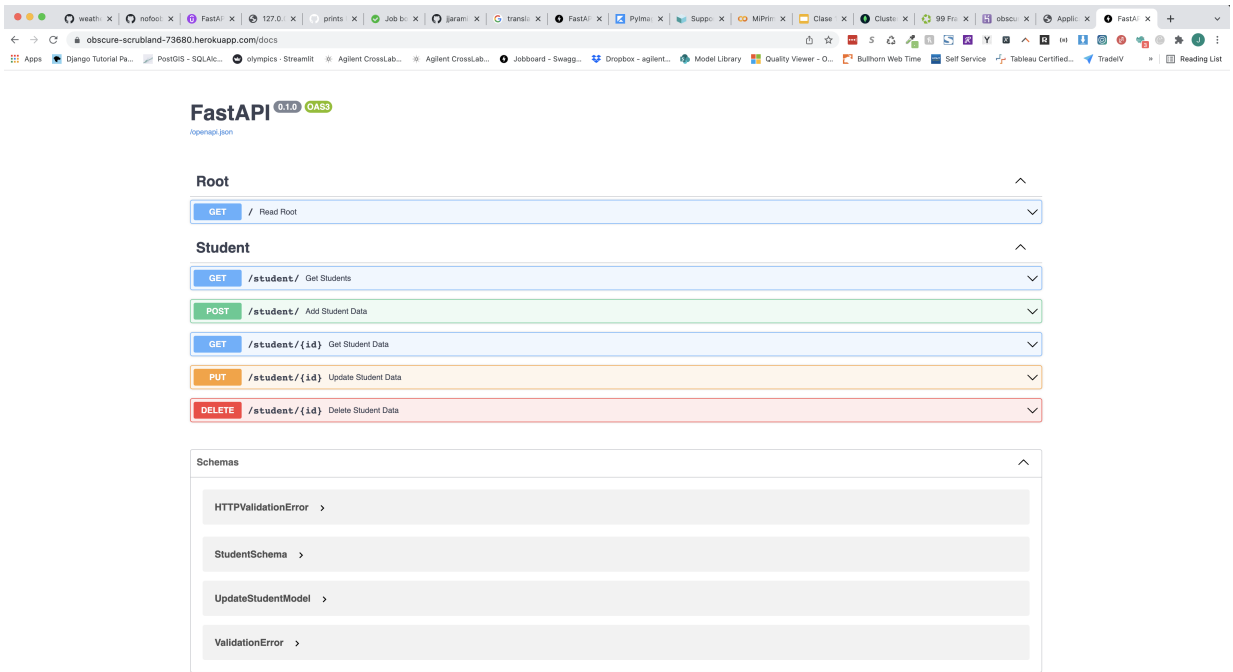
Nuevamente, asegúrese de reemplazaryour_connection_url con la URL de conexión real.

Envíe su código a Heroku y asegúrese de que se esté ejecutando al menos una instancia de la aplicación:

```
git push heroku master
heroku ps:scale web=1
```

Ejecute heroku open para abrir su aplicación en su navegador predeterminado.

```
heroku open
```



Ha implementado correctamente su aplicación en Heroku. Pruébela ahora si todas las funcionalidades CRUD trabajan.

Conclusión

En este tutorial, aprendió cómo crear una aplicación CRUD con FastAPI y MongoDB e implementarla en Heroku. Realice una autocomprobación rápida revisando los objetivos al comienzo del tutorial. Puede encontrar el código utilizado en este tutorial en GitHub.

¿Buscando por mas?

1. Configure pruebas unitarias y de integración con pytest.
2. Añadir rutas adicionales.
3. Create a GitHub repo for your application and configure CI/CD with GitHub Actions. En este blog encontraras mas sobre CI/CD GitHub Actions
4. Configure una IP estática en Heroku con Fixie Socks y restrinja el acceso a la base de datos MongoDB Atlas.