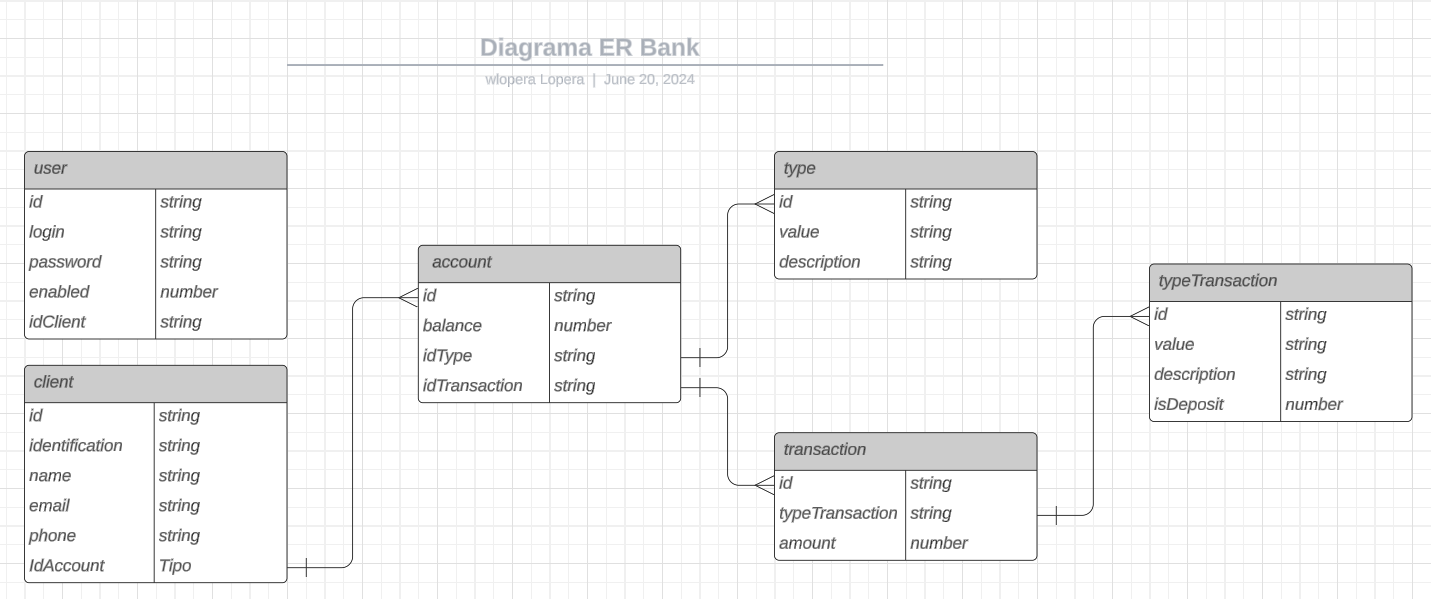
**PROYECTO BANCA PYTHON, REACT Y REACT NATIVE**

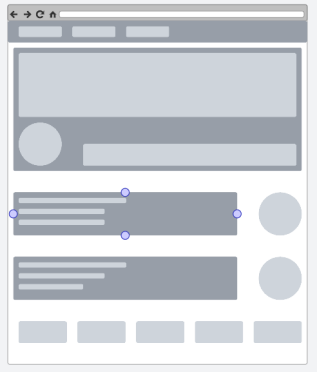
Proyecto de estudio de un Banco que permite conectarse como administrador y generar usuarios, cuentas y transacciones y como usuario permite conectarse, consultar cuentas y realizar transacciones (Pagos y transferencias bancarias).

* Python: APIs de servicios
  + - Usuarios
    - Clientes
    - Cuentas
    - Tipos
    - Transacciones
    - Tipo de transacciones

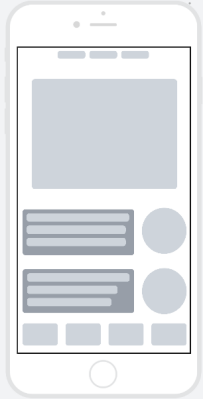


Nota: MongoDB

* React: Web del banco
  + - Admin: administrar usuarios y cuentas
    - Usuarios: Consulta y transacciones bancarias



* React Native: Consulta y transacciones bancarias solo para usuarios



**Python:**

* Desarrollo de API de usuarios que permiten ingresar a la banca. Inicialmente se va a permitir el acceso pero luego se debe generar token y validaciones de seguridad.

Librerías requeridas:

* + - FastApi (fastapi)
    - APIRouter (fastapi)
    - HTTPException (fastapi)
    - BaseModel (pydantic)
    - pymongo (MongoClient)
    - uvicorn Para levantar el servidor de python

pip install fastapi

<https://pypi.org/project/fastapi/>

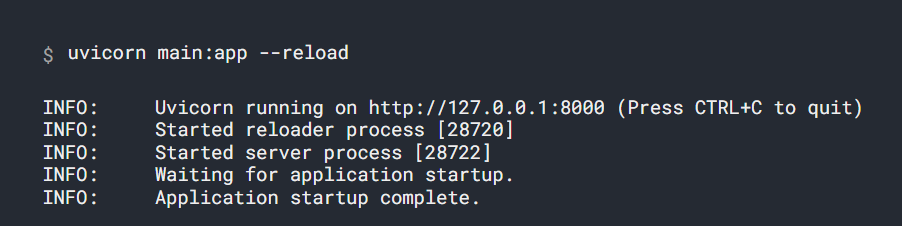
<https://fastapi.tiangolo.com/es/>

pip install "uvicorn[standard]"

<https://pypi.org/project/uvicorn/>

<https://pywombat.com/articles/introduccion-pydantic>

Para levantar servidor



pip install pymongo

<https://pypi.org/project/pymongo/>

Extensiones o plugin para VSCODE

* + - THUNDER CLINT Cliente para consumo de servicios API
    - Python
    - Python Extension Pack
* Probar Servidor de Python

from fastapi import FastAPI

app = FastAPI()

@app.get('/')

async def getUser():

    return {

        'id':"AQSWDE12345FRGTFG654646",

        'login':"wlopera",

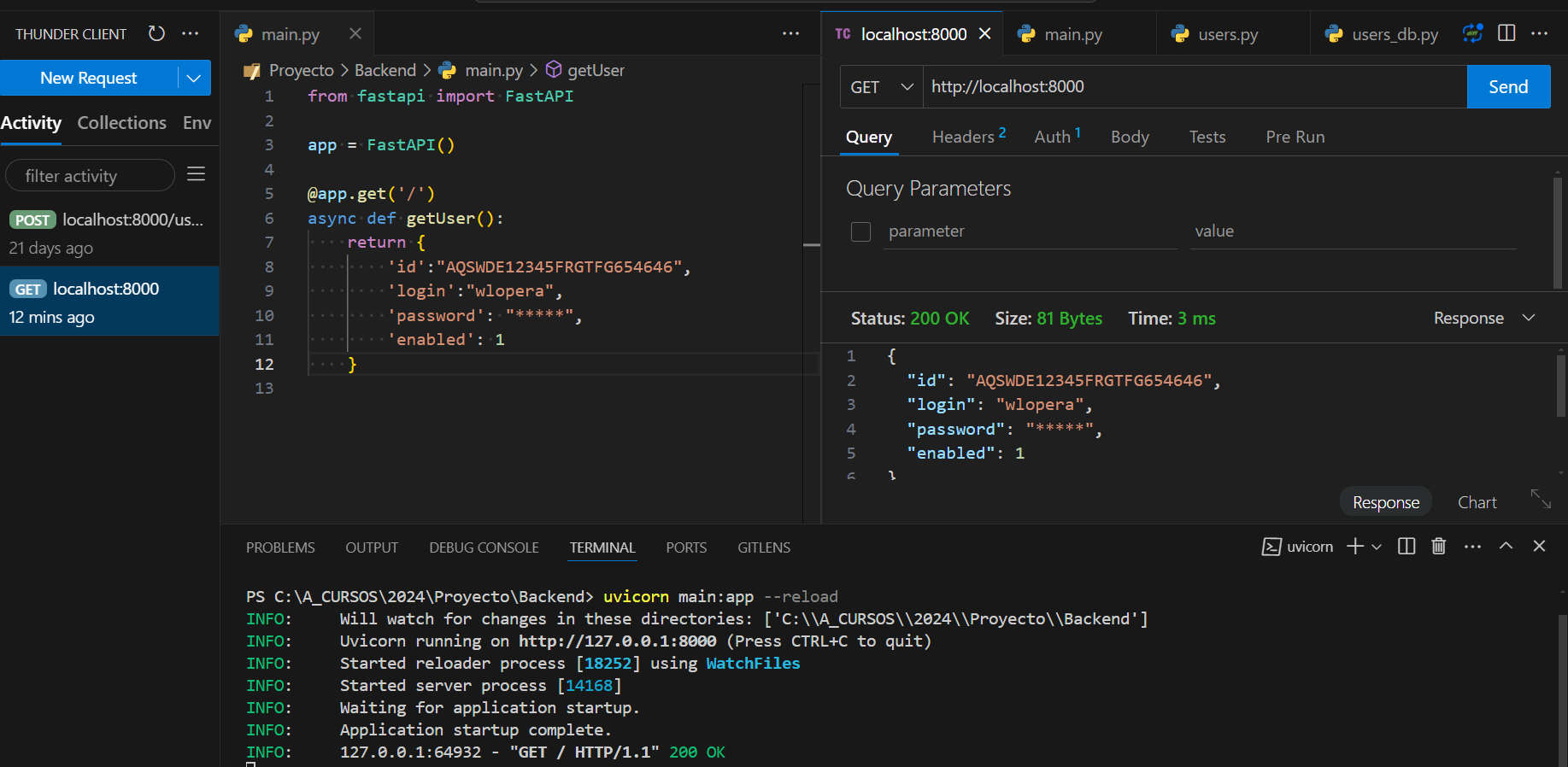
        'password': "\*\*\*\*\*",

        'enabled': 1

    }

Levantar y probar servicio

PS C:\A\_CURSOS\2024\Proyecto\Backend> uvicorn main:app --reload



Agregar route para manejo de varios APIS. Route API users

**users.py**

from fastapi import APIRouter

from pydantic import BaseModel

router = APIRouter(prefix="/users",

                   tags=["Usuarios"],

                   responses={404: {'message':"No Encontrado"}})

# Clase Usuario

class User(BaseModel):

    id: str

    login:str

    password:str

    enable: int

#Lista de Usuario

users\_list =[User(id="aqswe12345", login="lmessi",password="11111",enable=1),

             User(id="q1w2e3r4t5", login="cr7",password="22222",enable=1),

             User(id="f5g3h6j7hh", login="njunior",password="33333",enable=1),

             User(id="e3r4tf6ytf", login="lsuarez",password="44444",enable=0),

             User(id="98765fgdrs", login="jarango",password="55555",enable=1)]

# Consultar usuarios

@router.get("/")

async def users():

    return users\_list

**main.py**

from fastapi import FastAPI

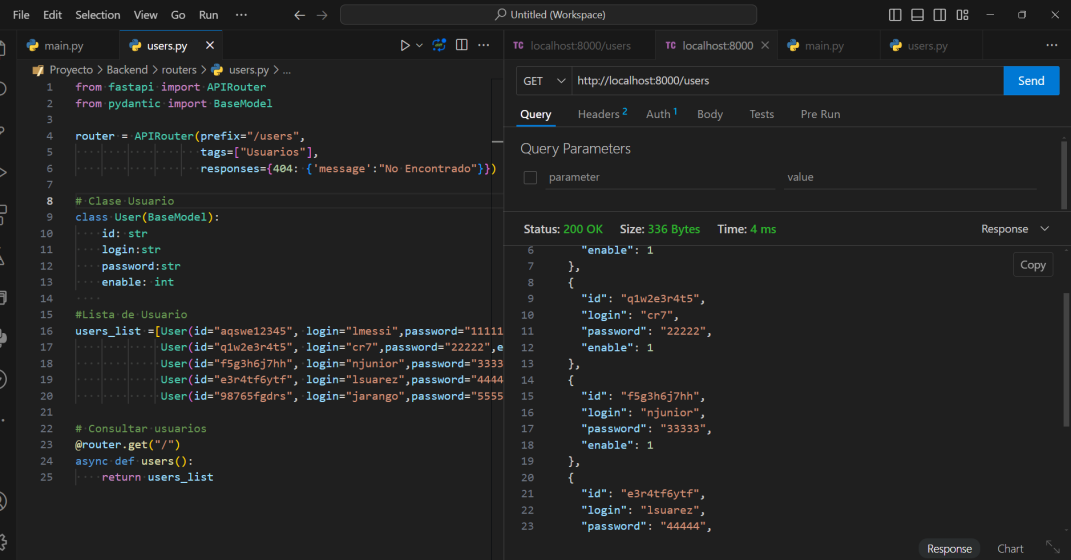
from routers import users

app = FastAPI()

# Routers

app.include\_router(users.router)

* Salida



**Uso de MongoDB**

* Instalar MongoDB y Herramientas de MongoDB:

<https://www.mongodb.com/docs/manual/installation>

<https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows>

* Agregar MongoDb al path
* Crear carpetas para db y log:

- C:\A\_CURSOS\2024\Proyecto\Backend\data\db

- C:\A\_CURSOS\2024\Proyecto\Backend\data\log

* Crear archivo para levantar MongoDB:

- C:\A\_CURSOS\2024\Proyecto\Backend\data\mongod.cfg

storage:

dbPath: C:\A\_CURSOS\2024\Proyecto\Backend\data\db

# Descomentar si quiero crear archivo de log:

# systemLog:

# destination: file

# path: C:\A\_CURSOS\2024\Proyecto\Backend\data\logs\mongod.log

# logAppend: true

net:

  bindIp: 127.0.0.1

  port: 27017

* Levantar MongoDB

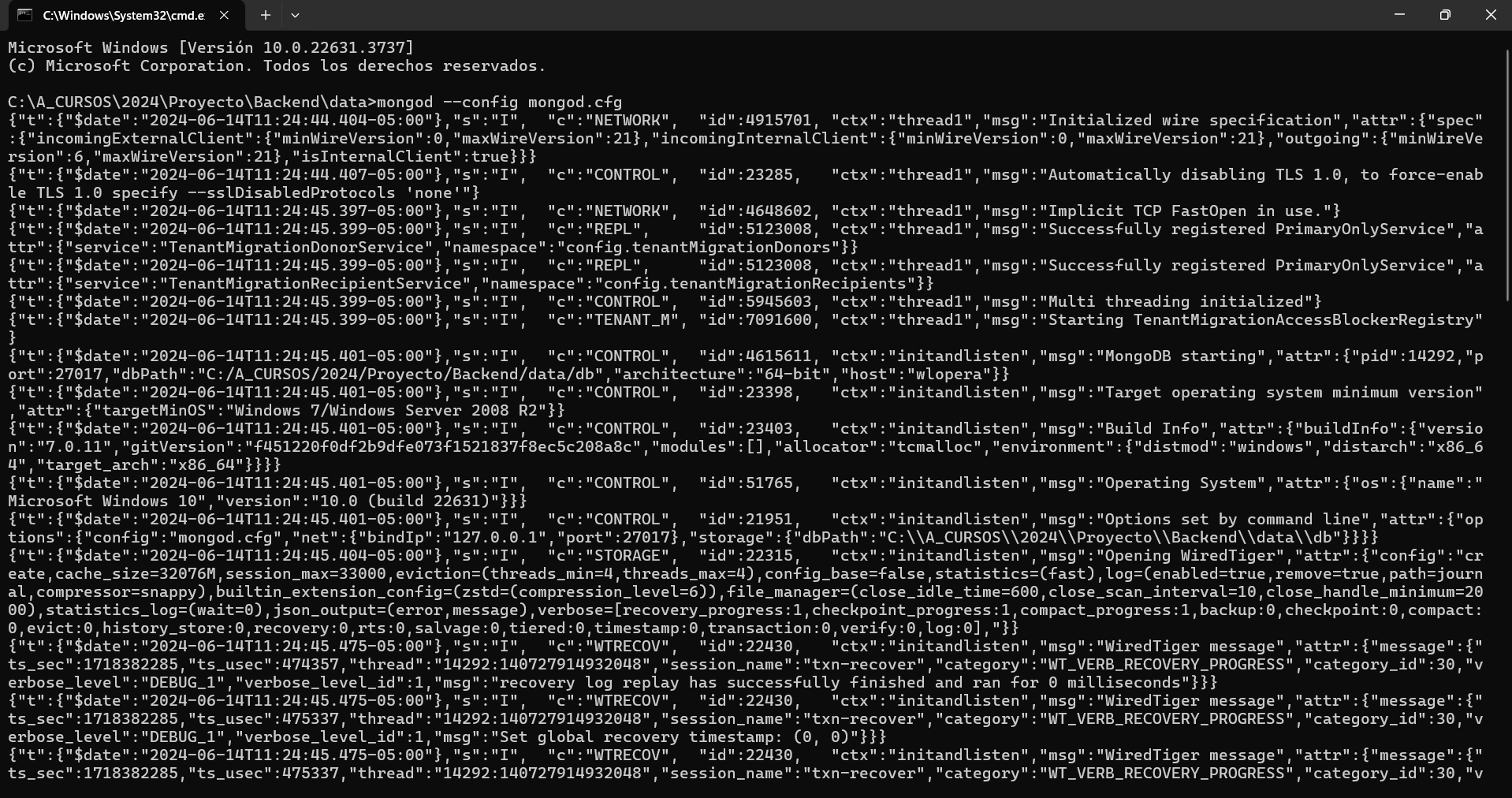
C:\A\_CURSOS\2024\Proyecto\Backend\data> **mongod --config mongod.cfg**

ó

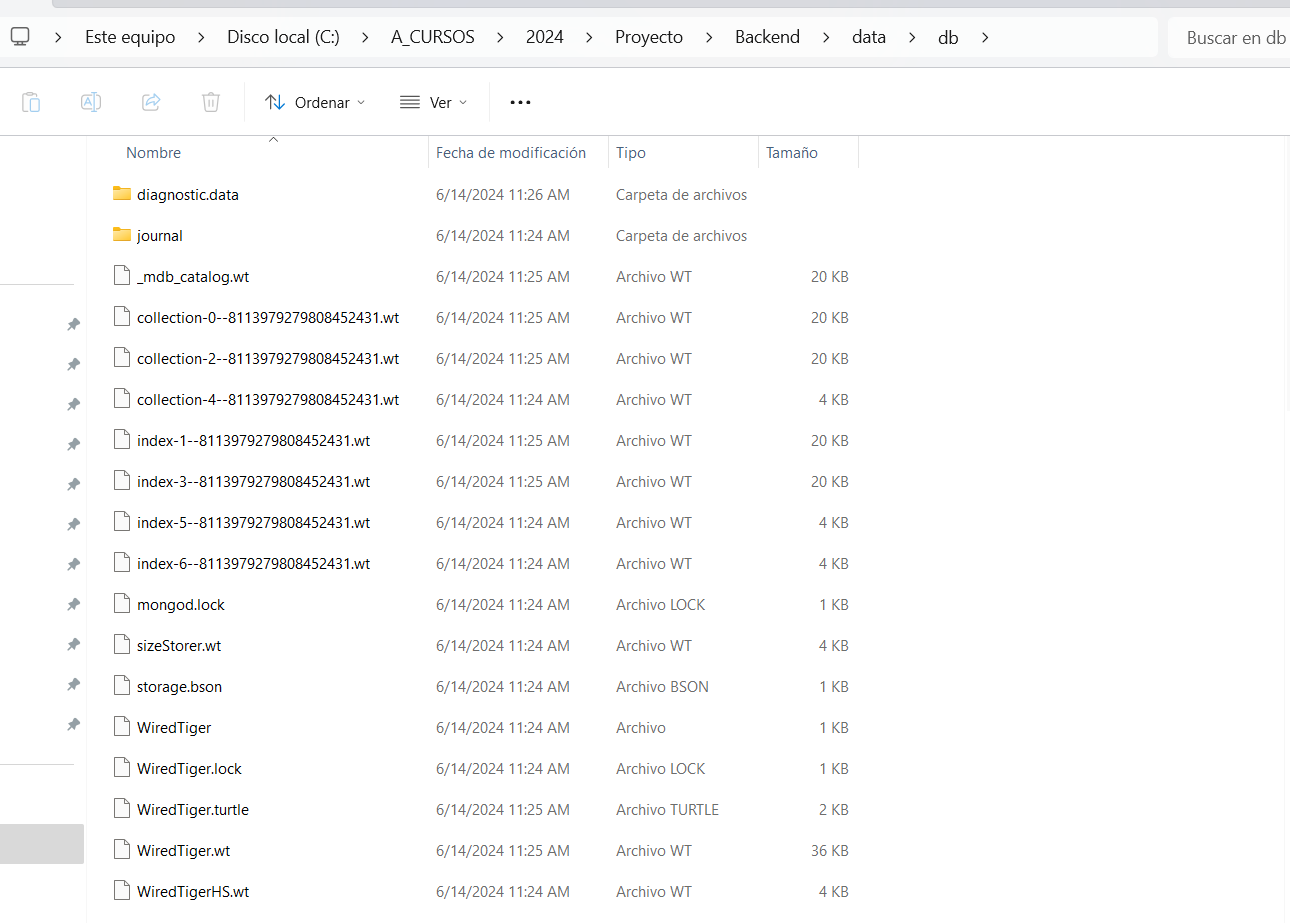
C:\A\_CURSOS\2024\Proyecto\Backend\data>

**mongod --dbpath "C:\A\_CURSOS\2024\Proyecto\Backend\data\db"**

Conexión: mongod://localhost [url:port]



Se crear el archivo de mongo en la ruta definida en el archivo config mongod.cfg



* Instalar Plugin MongoDB en Vscode

Conectar a MongoDB desde el plugin:

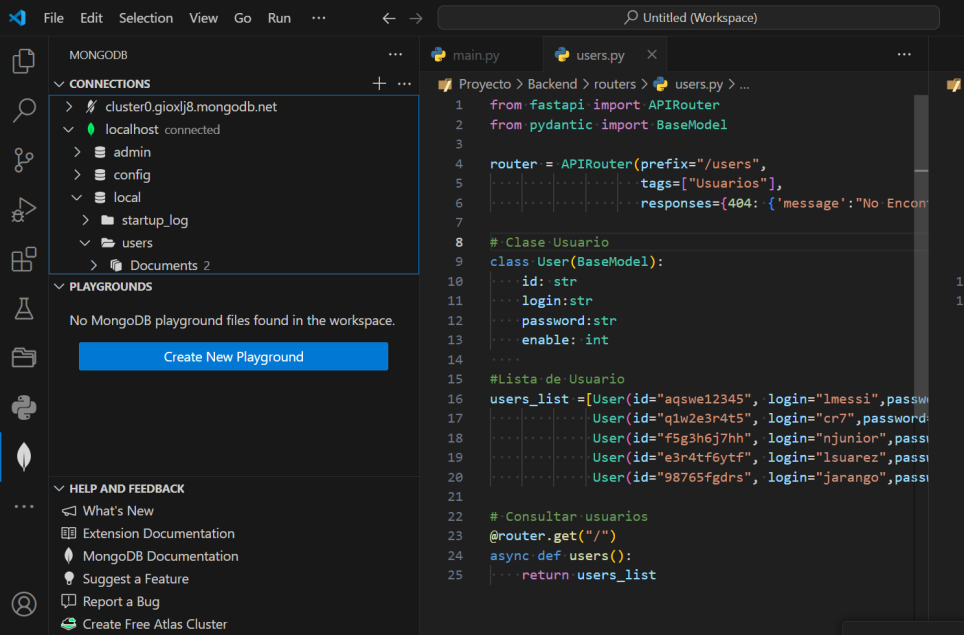
> mongodb://localhost:27017

localhost:

admin

config

local



* Conexio a MongoDB Atlas

- Crear Projecto en Mongo Atlas

- Crear DB an Mongo Atlas.

* Conectarse a Mongo Atlas desde VSCODE

\_> mongodb+srv://admin:<password>@cluster0.gioxlj8.mongodb.net/

\_>mongodb+srv://admin:admin@cluster0.gioxlj8.mongodb.net/

* Conectarse a Mongo Atlas desde Pyhton - 3.12+:

\_> mongodb+srv://admin:<password>@cluster0.gioxlj8.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0

\_> mongodb+srv://admin:admin@cluster0.gioxlj8.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0

* Codigo de Ejemplo:

from pymongo.mongo\_client import MongoClient

from pymongo.server\_api import ServerApi

uri = "mongodb+srv://admin:<password>@cluster0.gioxlj8.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"

# Create a new client and connect to the server

client = MongoClient(uri, server\_api=ServerApi('1'))

# Send a ping to confirm a successful connection

try:

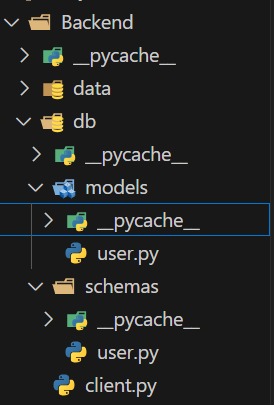
client.admin.command('ping')

print("Pinged your deployment. You successfully connected to MongoDB!")

except Exception as e:

print(e)

* Crear API de Usuarios conectado a MongoDB users\_db.py



Esquemas:

def user\_schemas(user) -> dict:

    return {

        "id": str(user["\_id"]),

        "login": user["login"],

        "password": user["password"],

        "enable": user["enable"]

    }

def users\_schemas(users) -> list:

    return [user\_schemas(user) for user in users]

Modelo:

### Clase usuario de negocio ###

from pydantic import BaseModel   # Permite crear una Entidad

from typing import Optional

class User(BaseModel):

    id: Optional[str] = None # Implica que es opcional

    login: str

    password: str

    enable: int

Cliente:

from pymongo import MongoClient

# Conexion a DB local (localhost)

db\_client = MongoClient().local

API

### USER DB API ###

from fastapi import APIRouter

from db.models.user import User

from db.schemas.user import user\_schemas, users\_schemas

from db.client import db\_client

from bson import ObjectId

router = APIRouter(prefix="/users",

                   tags=["Usuarios"],

                   responses={404: {'message':"No Encontrado"}})

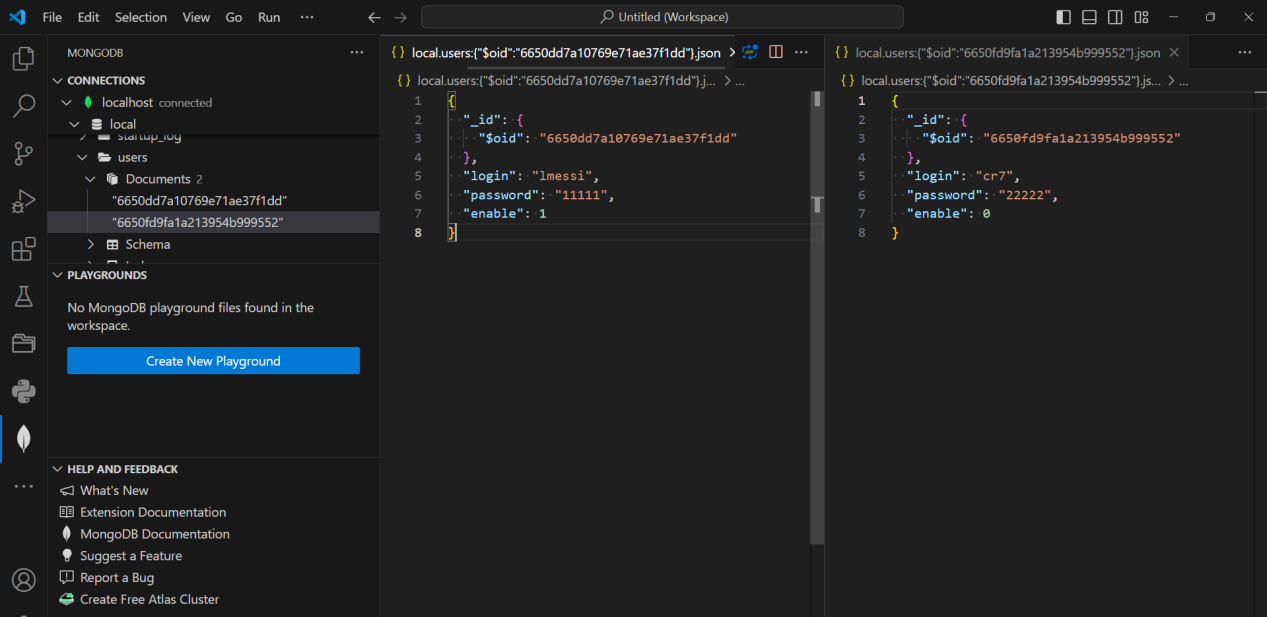
# API Consulta de usuarios

@router.get("/", response\_model=list[User])

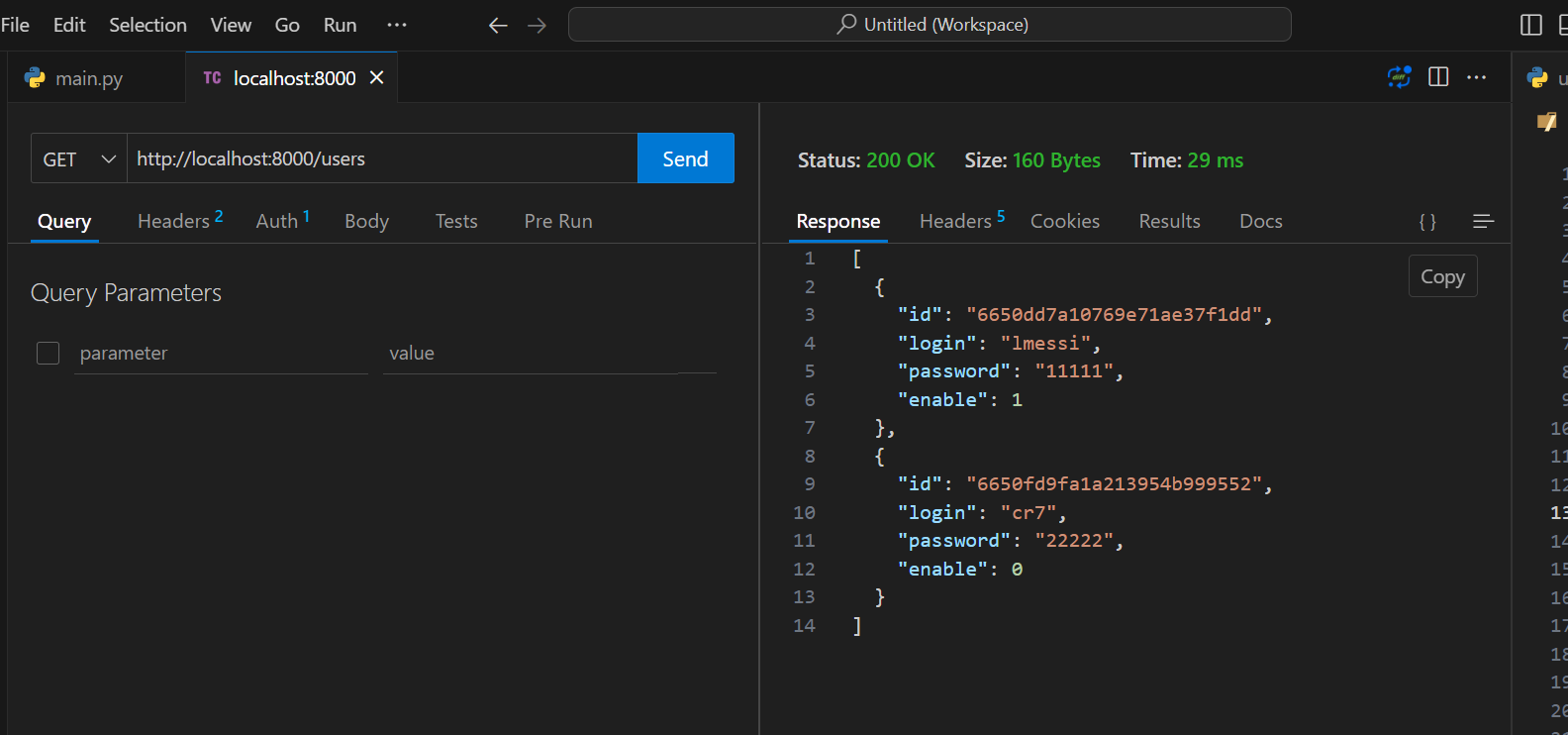
async def users():

    return users\_schemas(db\_client.users.find())

Data cargada como dummy en la DB de mongo:



Salida:



Consultas de MongoDB en Python. Algunas páginas de ayuda

<https://www.w3schools.com/python/python_mongodb_find.asp>

<https://www.w3schools.com/python/python_mongodb_query.asp>

user-db.py

...

class UserLogin(BaseModel):

    login: str

password: str

...

# API Consulta de usuario por login

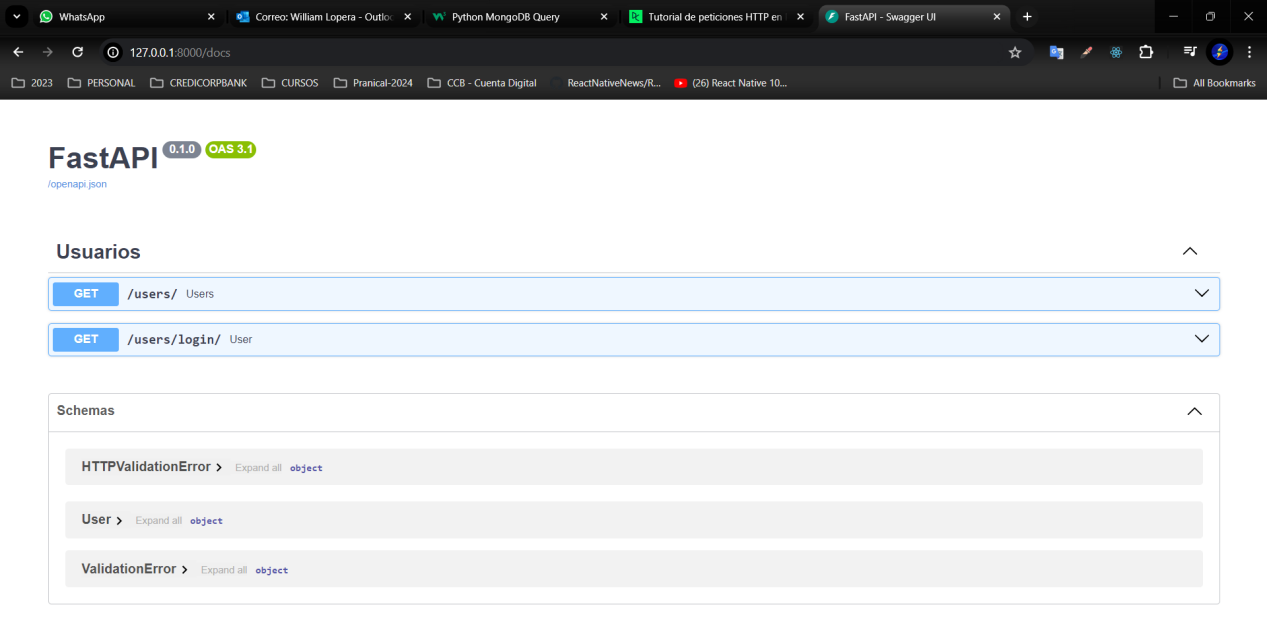
@router.get("/login/", response\_model=User)

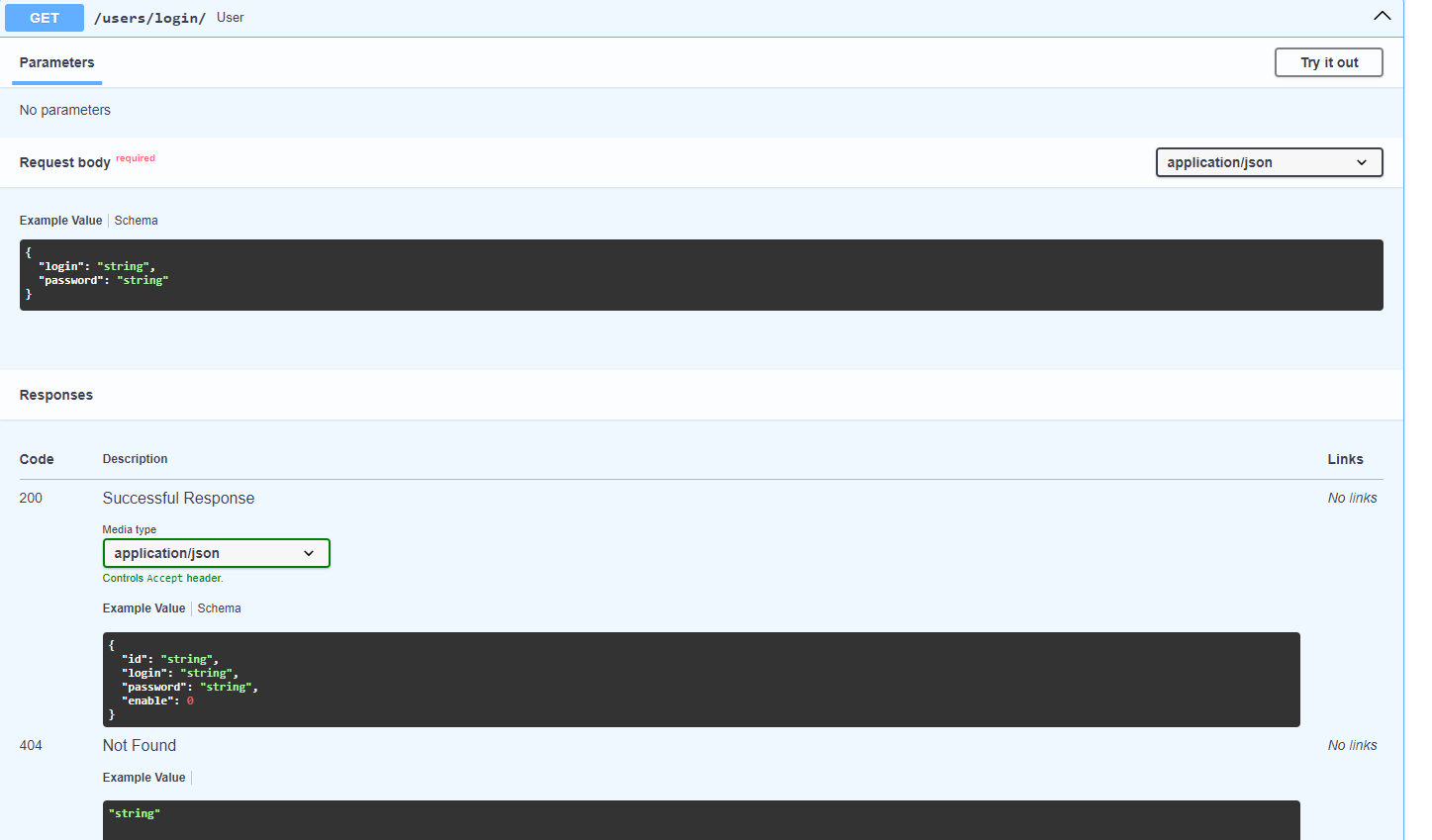
async def user(userLogin:UserLogin):

return user\_schemas(db\_client.users.find\_one({'login':userLogin.login}))

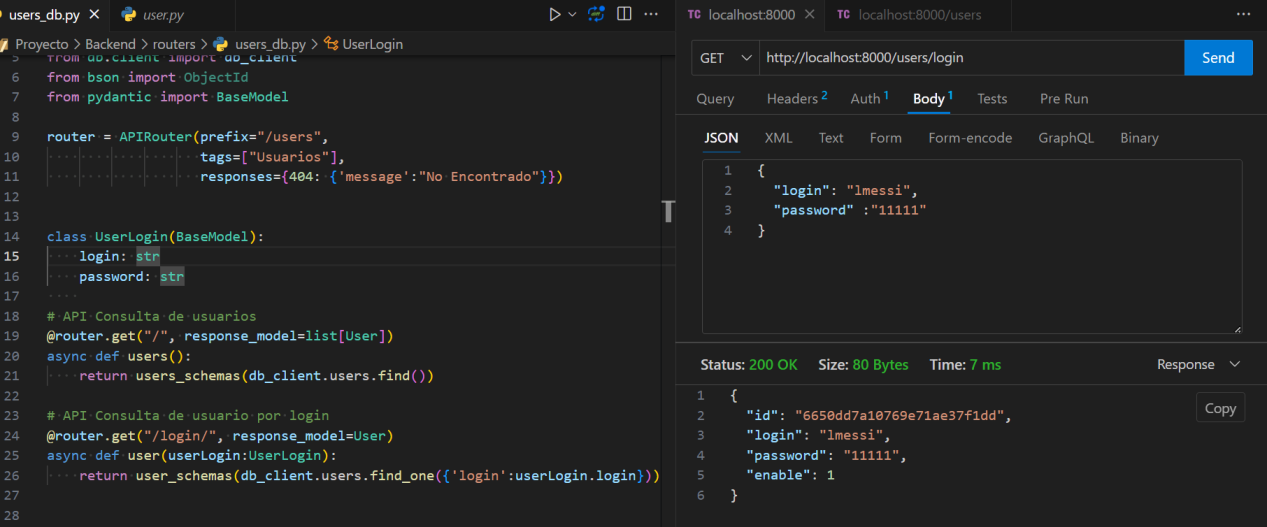
...

* Ver API vía web por Swagger





* Salida



**Código:**

* Clase Usuario

### Clase usuario de negocio ###

from pydantic import BaseModel, Field   # Permite crear una Entidad

from typing import Optional

from bson import ObjectId

class User\_db(BaseModel):

    \_id: Optional[ObjectId] = None

    login: str

    password: Optional[str] = None

    enabled: int

class User\_business(BaseModel):

    id: str

    login: str

    password: Optional[str] = None

enabled: int

* Esquema usuario

# Esquema de la DB User

def user\_schema(user, field) -> dict:

    return {

        field: str(user[field]),

        "login": user["login"],

        "password": user["password"],

        "enabled": user["enabled"]

    }

def users\_schema(users, field) -> list:

    return [user\_schema(user, field) for user in users]

* Route usuarios

### USER DB API ###

from fastapi import APIRouter, HTTPException, status

from db.models.user import User\_db, User\_business

from db.schemas.user import user\_schema, users\_schema

from db.client import db\_client

from bson import ObjectId

from pydantic import BaseModel

router = APIRouter(prefix="/users",

                   tags=["Usuarios"],

                   responses={404: {'message':"No Encontrado"}})

class UserLogin(BaseModel):

    login: str

    password: str

# API Consulta de usuarios

@router.get("/", response\_model=list[User\_business])

async def users():

    users\_db = users\_schema(db\_client.users.find(), "\_id")

    users\_business = [convert\_to\_user\_business(user) for user in users\_db]

    return users\_business

# API Consulta de usuario por login

@router.get("/login/", response\_model=User\_business)

async def user(userLogin:UserLogin):

    try:

        user\_db = db\_client.users.find\_one({"login": userLogin.login})

        user\_bussiness = convert\_to\_user\_business(user\_db)

        return user\_bussiness

    except:

        raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail={"error": "No se encontro usuario"})

# API activar o inactivar usuario

@router.put("/enabled/",  response\_model=User\_business)

async def setEnabled(user\_business:User\_business):

    try:

        update\_data = {

            'enabled': user\_business.enabled

        }

        db\_client.users.update\_one({"\_id": ObjectId(user\_business.id)},  {"$set": update\_data})

        return user\_business

    except:

        return "No se ha actualizado el estado del usuario"

# API modificar password

@router.put("/password/",  response\_model=User\_business)

async def setPassword(user\_business:User\_business):

    try:

        update\_data = {

            'password': user\_business.password

        }

        db\_client.users.update\_one({"\_id": ObjectId(user\_business.id)},  {"$set": update\_data})

        return user\_business

    except:

        return "No se ha actualizado el password"

# Crear usuario

@router.post("/add/", response\_model= User\_business, status\_code=status.HTTP\_201\_CREATED )

async def add\_user(user:User\_db):

    if type(search\_user("login", user.login)) == User\_db:

         raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="El usuario ya existe")

    else:

        # Crear usuario DB y convertir a Json

        user\_db = User\_db(login=user.login, password =user.password, enabled=user.enabled)

        # Esquema de usuario - generar usuario y obtener el id generado en mongoDB

        id = db\_client.users.insert\_one(dict(user)).inserted\_id

        # Consultar el usuario generado, directamente en mongoDB (\_id es como lo genera mongoDB)

        new\_user = db\_client.users.find\_one({"\_id": id})

        # Generar el objeto User de la respuesta del schema

        return convert\_to\_user\_business(new\_user)

# Funcion para consultar un usuario por campo generico

def search\_user(field: str, key):

    try:

        user\_db = db\_client.users.find\_one({field: key})

        return user\_db

    except:

        return {"error": "No se encontro usuario"}

# Función para convertir de User\_db a User\_business

def convert\_to\_user\_business(user\_db):

    return User\_business(

        id=str(user\_db['\_id']),

        login=user\_db['login'],

        # Comentar para no enviar al frontend

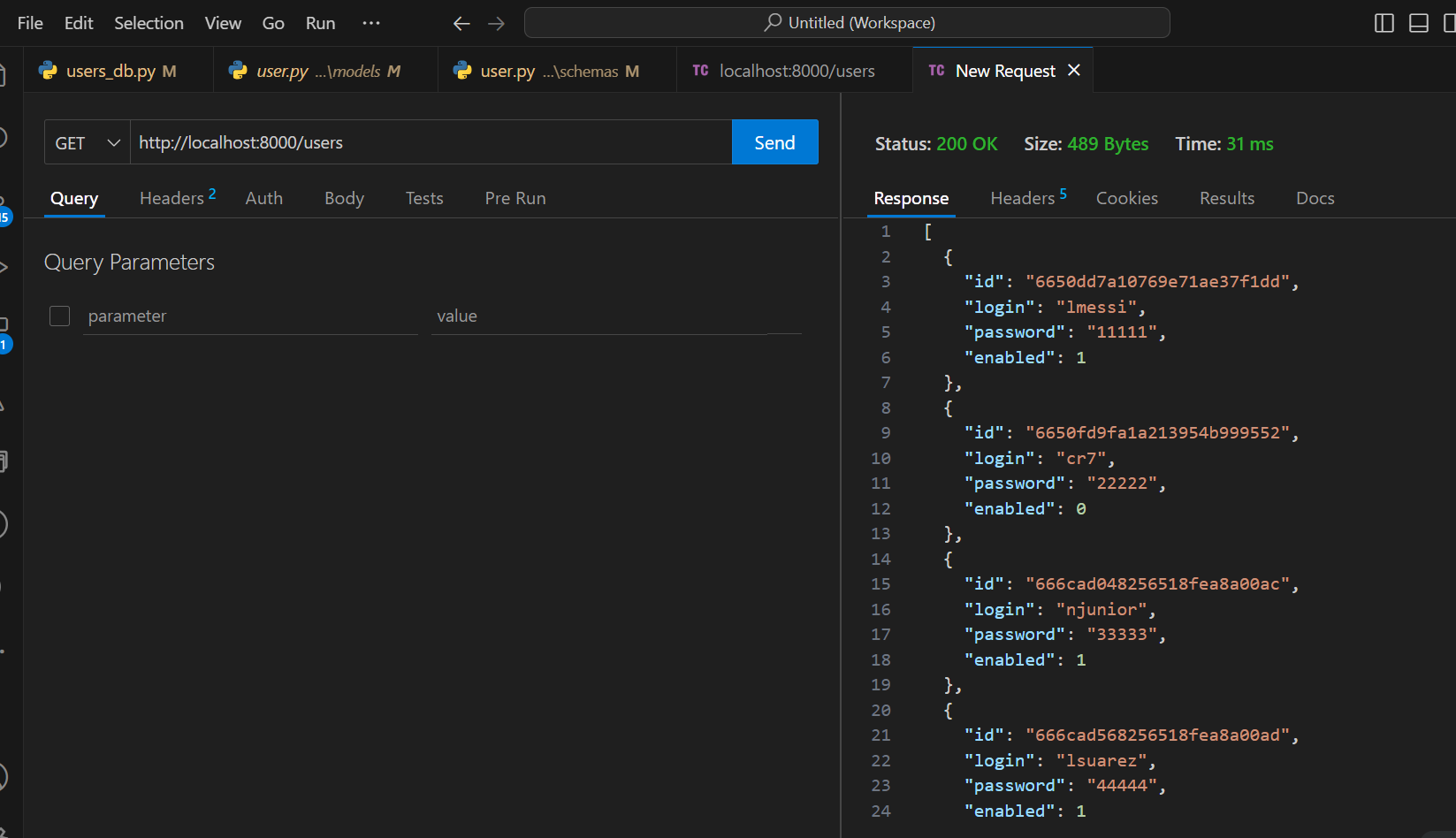
        password=user\_db['password'],

        enabled=user\_db['enabled']

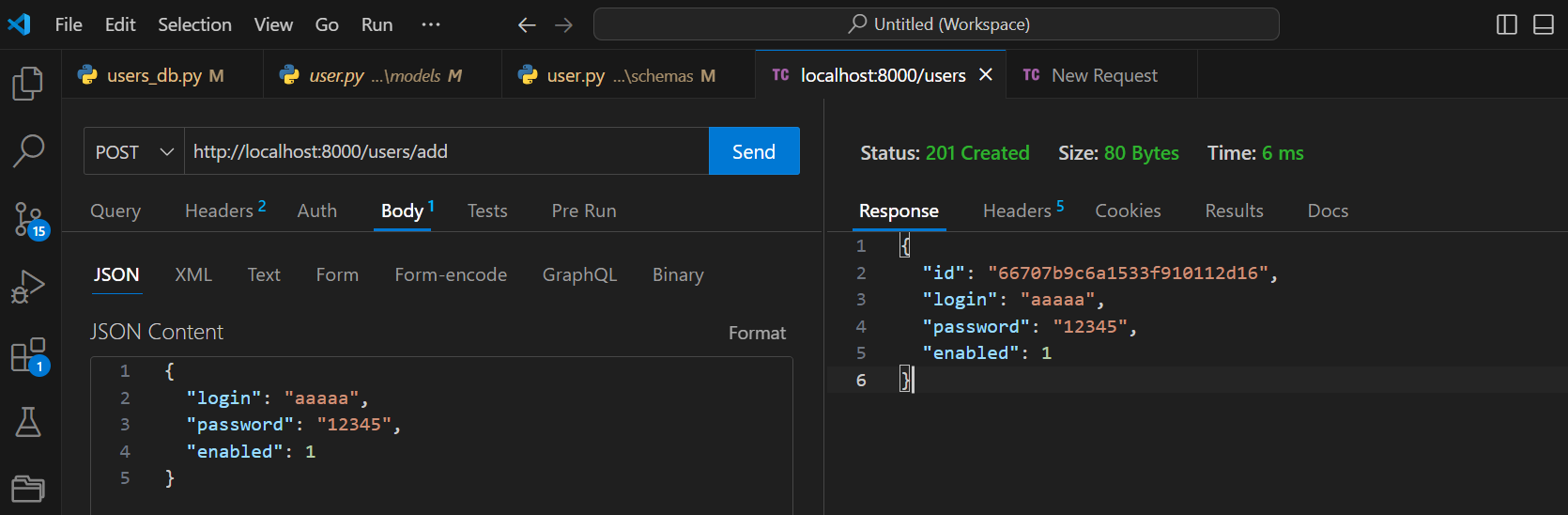
    )

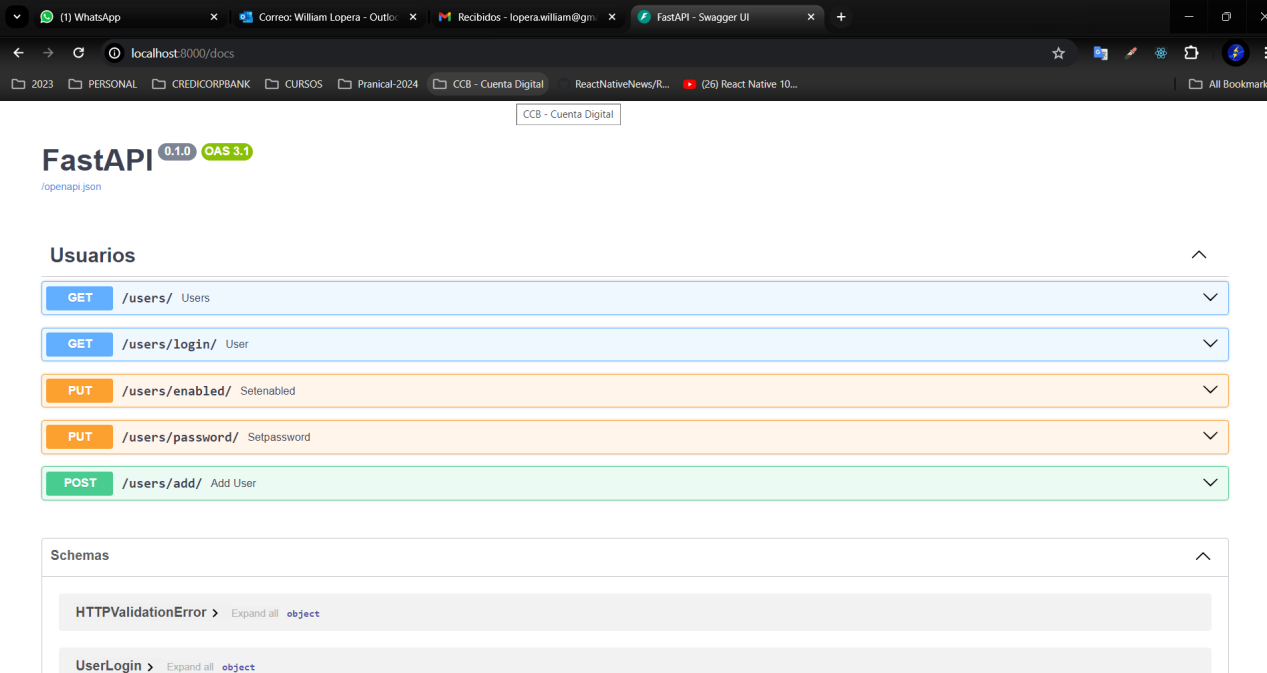
**Salida**

* Consultar Usuarios

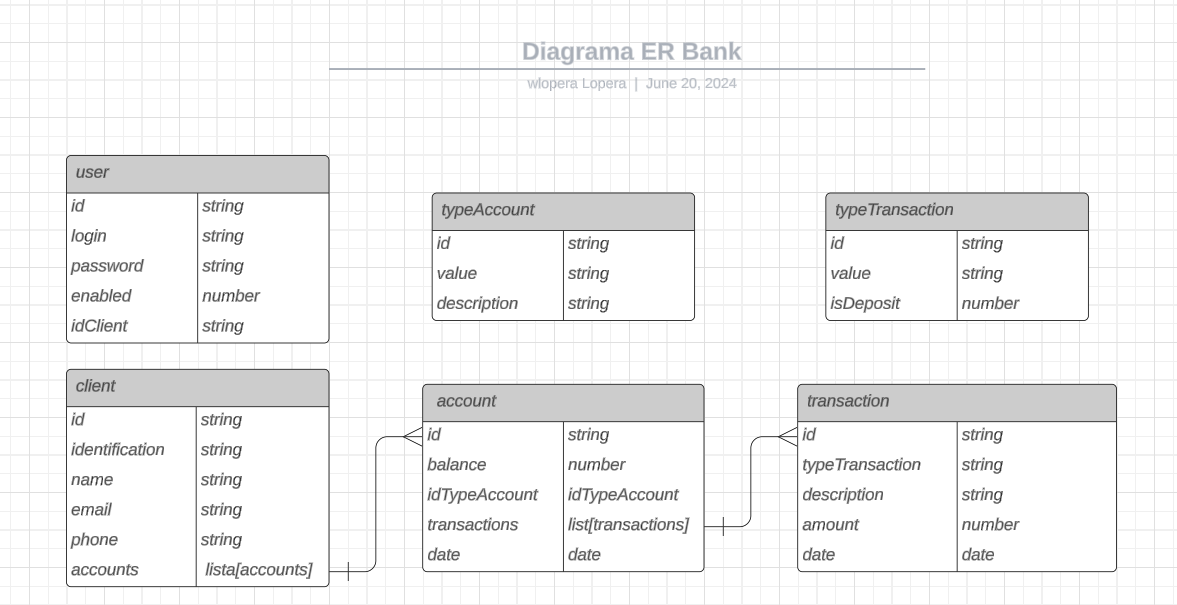


* Crear Usuario





**Nuevo modelo de Datos para MongoDB**



### **ANALISIS CHADGPT**

### Modelo de Datos para el Modelo No Embebido

#### Clientes (Modelo de Negocio y Consulta a la Base de Datos)

from pydantic import BaseModelfrom typing import Optional, List

class ClientDB(BaseModel):

\_id: Optional[str] = None

identification: str

name: str

email: str

phone: str

accounts: Optional[List[str]] = None

class ClientBusiness(BaseModel):

id: str

identification: str

name: str

email: str

phone: str

accounts: Optional[List[AccountDetails]] = None

class AccountDetails(BaseModel):

\_id: str

balance: float

idTypeAccount: str

transactions: Optional[List[str]] = None

date: str

#### Cuentas (Modelo de Negocio y Consulta a la Base de Datos)

class AccountDB(BaseModel):

\_id: Optional[str] = None

balance: float

idTypeAccount: str

transactions: Optional[List[str]] = None

date: str

class AccountBusiness(BaseModel):

id: str

balance: float

idTypeAccount: str

transactions: Optional[List[str]] = None

date: str

### Consulta y Armado del JSON de Respuesta

from fastapi import FastAPI, HTTPExceptionfrom pymongo import MongoClientfrom bson import ObjectId

# Conexión a MongoDB (ejemplo)

client = MongoClient('mongodb://localhost:27017/')

db = client['mydatabase']

clients\_collection = db['clients']

accounts\_collection = db['accounts']

app = FastAPI()

@app.get("/clients/{client\_id}/")async def get\_client\_with\_accounts(client\_id: str):

# Buscar el cliente por su \_id en la colección clients

client\_data = clients\_collection.find\_one({"\_id": ObjectId(client\_id)})

if client\_data is None:

raise HTTPException(status\_code=404, detail="Cliente no encontrado")

# Convertir el \_id a str para la respuesta

client\_data['\_id'] = str(client\_data['\_id'])

# Obtener detalles de las cuentas asociadas al cliente

accounts\_data = []

for account\_id in client\_data.get('accounts', []):

account = accounts\_collection.find\_one({"\_id": ObjectId(account\_id)})

if account:

account['\_id'] = str(account['\_id']) # Convertir ObjectId a str

accounts\_data.append(account)

# Construir la respuesta final

client\_business = ClientBusiness(\*\*client\_data, accounts=[])

client\_business.accounts = [AccountDetails(\*\*account) for account in accounts\_data]

return client\_business

### Ejemplo del JSON a Enviar al Frontend

Suponiendo que tenemos un cliente con \_id ObjectId 61465c73c81e2e0a4b5bb3a1 y dos cuentas asociadas con \_id ObjectId 61465c73c81e2e0a4b5bb3a2 y 61465c73c81e2e0a4b5bb3a3, el JSON de respuesta sería algo como esto:

{

"id": "61465c73c81e2e0a4b5bb3a1",

"identification": "123456789",

"name": "John Doe",

"email": "john.doe@example.com",

"phone": "1234567890",

"accounts": [

{

"\_id": "61465c73c81e2e0a4b5bb3a2",

"balance": 5000.0,

"idTypeAccount": "savings",

"transactions": [],

"date": "2023-05-01"

},

{

"\_id": "61465c73c81e2e0a4b5bb3a3",

"balance": 2000.0,

"idTypeAccount": "checking",

"transactions": [],

"date": "2023-06-15"

}

]}

### Explicación del Código y JSON de Respuesta

**Modelos de Datos**: Definimos modelos Pydantic para ClientDB, ClientBusiness, AccountDB, AccountBusiness, y AccountDetails para representar los datos del cliente y las cuentas.

**Consulta y Armado del JSON**: La función get\_client\_with\_accounts busca el cliente por su \_id, luego busca las cuentas asociadas y las convierte en instancias de AccountDetails. Luego construye una instancia de ClientBusiness que combina los datos del cliente con los detalles de sus cuentas.

**JSON de Respuesta**: La respuesta JSON contiene todos los datos del cliente y los detalles completos de sus cuentas asociadas.

Este enfoque te permite mantener una estructura limpia y clara en tu API FastAPI para manejar consultas a MongoDB utilizando referencias, asegurando que los datos enviados al frontend estén bien estructurados y sean fáciles de consumir. Ajusta según las necesidades específicas de tu aplicación y los requisitos del frontend.

**Programar solo clientes y cuentas**

* Clientes:

### Clase cliente de negocio y cliente DB ###

from pydantic import BaseModel

from typing import Optional, List

from bson import ObjectId

from models.account import Account\_bussiness

class Client\_db(BaseModel):

    \_id: Optional[ObjectId] = None

    identification: str

    name: str

    email:str

    phone: str

    accounts: Optional[List[str]] = None

class Client\_bussiness(BaseModel):

    id: str

    identification: str

    name: str

    email:str

    phone: str

    accounts: Optional[List[Account\_bussiness]] = None

* Cuentas:

### Clase cuentas de negocio y cuentas DB ###

from pydantic import BaseModel

from typing import Optional, List

from bson import ObjectId

class Account\_db(BaseModel):

    \_id: Optional[ObjectId] = None

    balance: float

    idTypeAccount: str

    transactions: Optional[List[str]] = None

    date: str

class Account\_bussiness(BaseModel):

    id: str

    balance: float

    idTypeAccount: str

    transactions: Optional[List[str]] = None

    date: str

* Vamos a crear datos en MongoDB para probar la consulta
* Creo la colección accounts

/\* global use, db \*/

// MongoDB Playground

// Use Ctrl+Space inside a snippet or a string literal to trigger completions.

const database = 'local';

const collection = 'accounts';

// The current database to use.

use(database);

// Create a new collection.

db.createCollection(collection);

// The prototype form to create a collection:

/\* db.createCollection( <name>,

  {

    capped: <boolean>,

    autoIndexId: <boolean>,

    size: <number>,

    max: <number>,

    storageEngine: <document>,

    validator: <document>,

    validationLevel: <string>,

    validationAction: <string>,

    indexOptionDefaults: <document>,

    viewOn: <string>,

    pipeline: <pipeline>,

    collation: <document>,

    writeConcern: <document>,

    timeseries: { // Added in MongoDB 5.0

      timeField: <string>, // required for time series collections

      metaField: <string>,

      granularity: <string>,

      bucketMaxSpanSeconds: <number>, // Added in MongoDB 6.3

      bucketRoundingSeconds: <number>, // Added in MongoDB 6.3

    },

    expireAfterSeconds: <number>,

    clusteredIndex: <document>, // Added in MongoDB 5.3

  }

)\*/

// More information on the `createCollection` command can be found at:

// https://www.mongodb.com/docs/manual/reference/method/db.createCollection/

* Creo dos registro de cuentas sin transacciones asociadas

// MongoDB Playground

// Use Ctrl+Space inside a snippet or a string literal to trigger completions.

// The current database to use.

use('local');

// Create a new document in the collection.

db.getCollection('accounts').insertOne({

    balance: 100.00,

    idTypeAccount: '61465c73c81e2e0a4b5bb3a2', # Cuenta de Ahorro

    date: '2024-05-18T14:23:45'

});

// MongoDB Playground

// Use Ctrl+Space inside a snippet or a string literal to trigger completions.

// The current database to use.

use('local');

// Create a new document in the collection.

db.getCollection('accounts').insertOne({

    balance: 1000.00,

    idTypeAccount: '61465c73c81e2e0a4b5bb3a3', # Cuenta Corriente

    date: '2024-04-05T10:25:00'

});

Cuentas Creadas:

[

  {

    "\_id": "66747cdfb238ea28ed4da42d",

    "balance": 100,

    "idTypeAccount": "61465c73c81e2e0a4b5bb3a2",

    "date": "2024-05-18T14:23:45"

  },

  {

    "\_id": "66747d47b9defe6d58e83c55",

    "balance": 1000,

    "idTypeAccount": "61465c73c81e2e0a4b5bb3a3",

    "date": "2024-04-05T10:25:00"

  }

]

* + Creo la colección clients

/\* global use, db \*/

// MongoDB Playground

// Use Ctrl+Space inside a snippet or a string literal to trigger completions.

const database = 'local';

const collection = 'clients';

// The current database to use.

use(database);

// Create a new collection.

db.createCollection(collection);

// The prototype form to create a collection:

/\* db.createCollection( <name>,

  {

    capped: <boolean>,

    autoIndexId: <boolean>,

    size: <number>,

    max: <number>,

    storageEngine: <document>,

    validator: <document>,

    validationLevel: <string>,

    validationAction: <string>,

    indexOptionDefaults: <document>,

    viewOn: <string>,

    pipeline: <pipeline>,

    collation: <document>,

    writeConcern: <document>,

    timeseries: { // Added in MongoDB 5.0

      timeField: <string>, // required for time series collections

      metaField: <string>,

      granularity: <string>,

      bucketMaxSpanSeconds: <number>, // Added in MongoDB 6.3

      bucketRoundingSeconds: <number>, // Added in MongoDB 6.3

    },

    expireAfterSeconds: <number>,

    clusteredIndex: <document>, // Added in MongoDB 5.3

  }

)\*/

// More information on the `createCollection` command can be found at:

// https://www.mongodb.com/docs/manual/reference/method/db.createCollection/

* Creo un registro de cliente asociando las dos cuentas anteriores

// MongoDB Playground

// Use Ctrl+Space inside a snippet or a string literal to trigger completions.

// The current database to use.

use('local');

// Create a new document in the collection.

db.getCollection('clients').insertOne({

    identification: "1234567-1-123456",

    name: "William Lopera",

    email:"lopera.william@test.com",

    phone: "+507 6432 1404",

    accounts: [

        ObjectId("66747cdfb238ea28ed4da42d"),

        ObjectId("66747d47b9defe6d58e83c55")

    ]

});

Cliente Creado:

{

  "\_id": {

    "$oid": "66747e3fa029368fe79c602a"

  },

  "identification": "1234567-1-123456",

  "name": "William Lopera",

  "email": "lopera.william@test.com",

  "phone": "+507 6432 1404",

  "accounts": [

    {

      "$oid": "66747cdfb238ea28ed4da42d"

    },

    {

      "$oid": "66747d47b9defe6d58e83c55"

    }

  ]

}

**Python para consultar el cliente y sus cuentas asociadas**

client.py

### Clase cliente de negocio y cliente DB ###

from pydantic import BaseModel

from typing import Optional, List

from bson import ObjectId

from db.models.account import Account

"""

    Clase: Clientes de base de datos

    wlopera

    @Jun 2024

"""

class Client\_db(BaseModel):

    \_id: Optional[ObjectId] = None

    identification: str

    name: str

    email:str

    phone: str

    accounts: Optional[List[str]] = None

"""

    Clase: Clientes simple (con id de las cuentas)

    wlopera

    @Jun 2024

"""

class Client(BaseModel):

    id: str

    identification: str

    name: str

    email:str

    phone: str

    accounts: Optional[List[str]] = None

"""

    Clase: Clientes completos (con datos de las cuentas)

    wlopera

    @Jun 2024

"""

class Client\_full(BaseModel):

    id: str

    identification: str

    name: str

    email:str

    phone: str

    accounts: Optional[List[Account]] = None

account.py

### Clase cuentas de negocio y cuentas DB ###

from pydantic import BaseModel

from typing import Optional, List

from bson import ObjectId

"""

    Clase: Cuentas del cliente en base de datos

    wlopera

    @Jun 2024

"""

class Account\_db(BaseModel):

    \_id: Optional[ObjectId] = None

    balance: float

    idTypeAccount: str

    transactions: Optional[List[str]] = None

    date: str

"""

    Clase: Cuentas del cliente (negocio)

    wlopera

    @Jun 2024

"""

class Account(BaseModel):

    id: str

    balance: float

    idTypeAccount: str

    transactions: Optional[List[str]] = None

    date: str

client\_db.py

### CLIENTES DB API ###

from fastapi import APIRouter, HTTPException, status

from db.models.client import Client, Client\_full

from db.models.account import Account

from db.client import db\_client

from bson import ObjectId

router = APIRouter(

    prefix='/clients',

    tags=["Clientes"],

    responses={status.HTTP\_404\_NOT\_FOUND: {'message':"No encontrado"}}

)

"""

    API: Clientes simple (con id de las cuentas)

    wlopera

    @Jun 2024

"""

@router.get('/', response\_model=list[Client])

async def getClients():

    clients\_db = db\_client.clients.find()

    clients = []

    for client\_db in clients\_db:

        account = []

        for account\_id in client\_db.get('accounts', []):

            account.append(str(account\_id))

        clients.append(create\_client(client\_db, account))

return clients

"""

    API: Clientes completa (con datos de las cuentas)

    wlopera

    @Jun 2024

"""

@router.get('/full/', response\_model=list[Client\_full])

async def getClientsFull():

    clients\_db = db\_client.clients.find()

    clients = []

    for client\_db in clients\_db:

        accounts = []

        for account\_id in client\_db.get('accounts', []):

            account\_db = db\_client.accounts.find\_one({"\_id": account\_id})

            accounts.append(create\_account(account\_db))

        clients.append(create\_client\_full(client\_db, accounts))

return clients

"""

    Metodo: Crear objeto cliente simple

    client\_db(Client\_db): Datos del cliente de base de datos

    @accounts(list[str]): Lista de id de cuentas del cliente (negocio)

    wlopera

    @Jun 2024

    return  Datos simple del cliente (negocio)

"""

def create\_client(client\_db, accounts):

    return Client(

        id= str(client\_db['\_id']),

        identification= client\_db['identification'],

        name= client\_db['name'],

        email= client\_db['email'],

        phone= client\_db['phone'],

        accounts= accounts

)

"""

    Meotodo: Crear objeto cliente completa

    client\_db(Client\_db): Datos del cliente de base de datos

    @accounts(list[Account]): Lista de  cuentas del cliente (negocio)

    wlopera

    @Jun 2024

    return  Datos completos del cliente (negocio)

"""

def create\_client\_full(client\_db, accounts):

    return Client\_full(

        id= str(client\_db['\_id']),

        identification= client\_db['identification'],

        name= client\_db['name'],

        email= client\_db['email'],

        phone= client\_db['phone'],

        accounts= accounts

    )

"""

    Crear objeto cuentas

    account\_db(Account\_db): Datos de la cuenta de base de datos

    wlopera

    @Jun 2024

    return  Datos de la cuenta del cliente (negocio)

"""

def create\_account(account\_db):

    return Account(

        id= str(account\_db['\_id']),

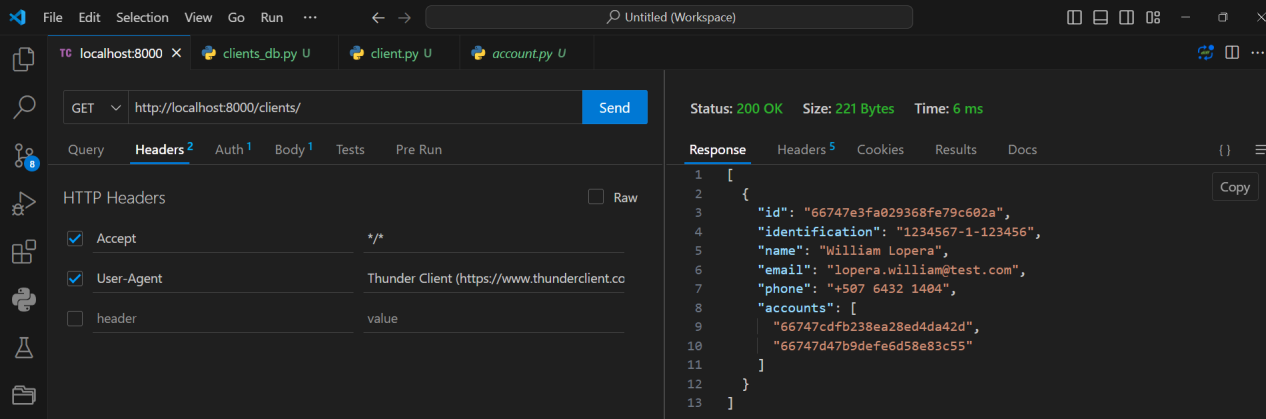
        balance= account\_db['balance'],

        idTypeAccount= account\_db['idTypeAccount'],

        date= account\_db['date'],

    )

Consulta de cliente simple



Consulta de cliente completa

