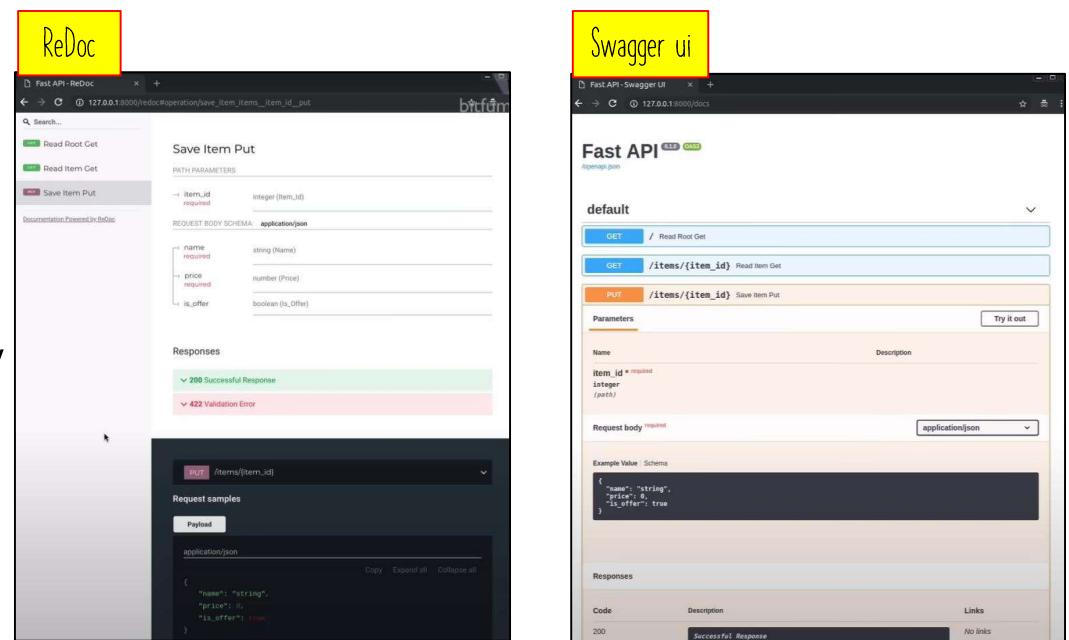


FAST API

Guizani Ahmed

FAST API

- A super fast Python web framework: very modern and nice features of synchronous programming which is still **lacking in django framework**.
- Fast API features:
 - **Automatic documentations:**
 - **swagger UI:** to check what are the routes we have created or api endpoints created or try them out (check what you are creating how it is going to respond).
 - **ReDoc:** same as swagger UI but more designed.
 - Uses python 3.6 and more: use also the pidantic library (provide type system like type hint variable such str or int...)



FAST API

- Fast API features (continious):
 - **Totaly based in open API standards and json schema:**
 - **open API:** define how to create your api under linux foundation.
 - **Json schema:** by default return json which every api modern need to communicate with other things.
 - Code editors **auto-complete** in VScode or pycharm.
 - **Security and authentication** (http basic security, Oauth 2 with JWT token, API keys in header, query parameters, cookies).
 - **Dependancy injection unlimited plugins testing:** it uses **pytest** to test your applications.
 - **Starlette Features:** another framework for python:
 - Websocket support
 - GraphQL support
 - In-process background tasks
 - Startup and shutdown events
 - Test client built on requests
 - CORS, gzip, Static files, Streaming responses
 - Session and Cookie support
 - Other Supports: SQL databases, nosql database.

INSTALL AND SETUP

- [GitHub - tiangolo/fastapi](#): FastAPI framework, high performance, easy to learn, fast to code, ready for production
- [FastAPI \(tiangolo.com\)](#)

1- create virtual environment with python3 **venv**:

```
C:\Users\Ahmed>python -m venv fastapi-env
```

2- activate the venv:

```
C:\Users\Ahmed>fastapi-env\scripts\activate  
(fastapi-env) C:\Users\Ahmed>
```

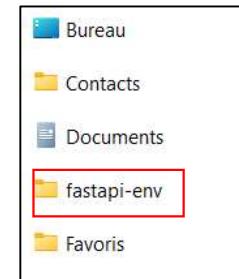
Rq ([link for syntaxe](#): `venv` – Creation of virtual environments – Python 3.12.1 documentation)

3- Install fast api (installation dans le venv seulement):

```
(fastapi-env) C:\Users\Ahmed>pip3 install fastapi  
Collecting fastapi  
  Obtaining dependency information for fastapi from http://pypi.org/simple/fastapi/  
    Downloading fastapi-0.108.0-py3-none-any.whl.metadata  
    Downloading fastapi-0.108.0-py3-none-any.whl.metadata
```

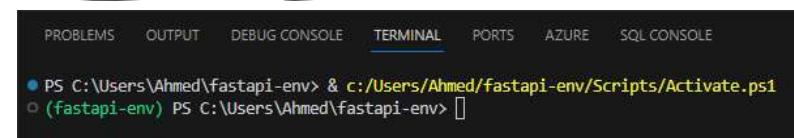
4- installation of ASGI server for production (Uvicorn or Hypercorn) only installed in virtual environment.

```
(fastapi-env) C:\Users\Ahmed>pip3 install uvicorn  
Collecting uvicorn  
  Downloading uvicorn-0.25.0-py3-none-any.whl.metadata (6.4 kB)  
Collecting click>=7.0 (from uvicorn)  
  Downloading click-8.1.7-py3-none-any.whl.metadata (3.0 kB)  
Collecting h11>=0.8 (from uvicorn)
```



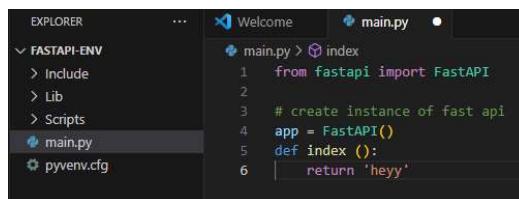
USES

5- in vscode after opening the folder fast api location: the activation of the venv run automatically:



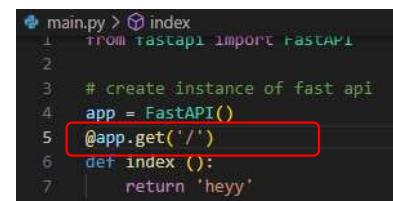
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE SQL CONSOLE
● PS C:\Users\Ahmed\fastapi-env> & c:/Users/Ahmed/fastapi-env/Scripts/Activate.ps1
○ (fastapi-env) PS C:\Users\Ahmed\fastapi-env>
```

6- now create the file main.py in vscode in the specific folder fastapi-env:



```
EXPLORER ... Welcome main.py
FASTAPI-ENV
> Include
> Lib
> Scripts
main.py
pyvenv.cfg
```

```
main.py > index
1 from fastapi import FastAPI
2
3 # create instance of fast api
4 app = FastAPI()
5 def index () :
6     return 'heyy'
```



```
main.py > index
1 from fastapi import FastAPI
2
3 # create instance of fast api
4 app = FastAPI()
5 @app.get('/')
6 def index () :
7     return 'heyy'
```

7- handle the path for the file to be executed from the server: `@app.get('/')`
/: pour indiquer que defalt path is localhost

8- run the server in vs code terminal in the venv: `uvicorn main:app --reload`
`--reload`: to tell the server to reload automatically when we change the file.

```
(fastapi-env) PS C:\Users\Ahmed\fastapi-env> uvicorn main:app --reload
```

9- dans le navigateur `127.0.0.1:8000`



USES

10- dictionnary use the return json format:

```
# create instance of fast api
app = FastAPI()
# decorate
@app.get('/')
# function
def index():
    #return 'comment va tu fast api'
    return {'data': {'name': 'Sarthak'}}
```

127.0.0.1:8000

```
{ "data": { "name": "Sarthak" }}
```

11- to add another path file to be use:

```
@app.get('/about')
def about():
    return {'data': {'about page'}}
```

127.0.0.1:8000/about

```
{ "data": [ "about page" ] }
```

12- initialise a Git repository: in terminal in venv

13- to use a blog you need the next syntaxe for routing easly with fastapi (Depending on the input of user, it will return the corresponding page to string as default)

```
@app.get('/blog/{id}')
def about(id):
    return {'data': id}
```

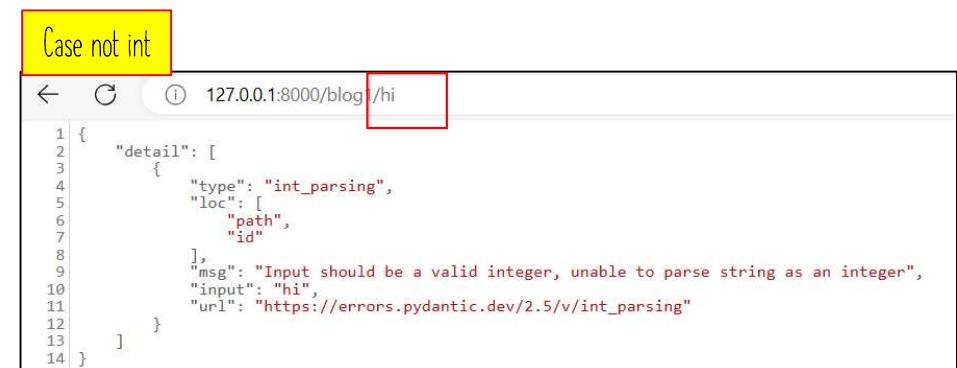
127.0.0.1:8000/blog/300

```
{ "data": "300" }
```

USES

- 13- if the entered value is an integer you need to add the int

```
# for specification of type hints
@app.get('/blog1/{id}')
def about(id: int):
    return {'data': id}
```



- 14- /blog/unpublished won't work since dynamic rooting above which is {id} to correct it we need to make all the function of unpublished above the dynamic {id}.

```
@app.get('/blog/{id}')
def show(id: int):
    # fetch blog with id = id
    return {'data': id}

@app.get('/blog/unpublished')
def unpublished():
    ...return {'data': 'all unpublished blogs'}
```

API DOCS - SWAGGER/REDOCS

- To acces to the feature of documentations we need to put **Localhost:8000/docs**

The screenshot shows the left sidebar of a FastAPI documentation page. It includes sections for 'default' and 'Schemas'. The 'default' section lists several API endpoints:

- GET / index
- GET /about About
- GET /blog/{id} About
- GET /blog1/{id} About

The 'Schemas' section lists two validation errors:

- HTTPValidationError > `detail: str`
- ValidationError > `detail: str`

The screenshot shows the Redoc interface for a specific endpoint: `GET /blog/{id}`. The 'Parameters' section shows a required path parameter `id` with the value `1566`, which is highlighted with a red box. Below the parameters is a 'Responses' section. The 'curl' command for generating a request is shown in a code block, also highlighted with a red box:

```
curl -X 'GET' \
'http://127.0.0.1:8000/blog/1566' \
-H 'accept: application/json'
```

The 'Request URL' is listed as `http://127.0.0.1:8000/blog/1566`. The 'Server response' section shows a successful 200 status code response. The 'Response body' is a JSON object:

```
{ "data": "1566" }
```

The 'Content-Type' header is listed as `application/json`. The 'Date' header is listed as `Thu, 04 Jan 2024 16:04:50 GMT`. The 'Server' header is listed as `uvicorn`.

API DOCS - SWAGGER/REDOCS

- To acces to the feature of documentations we need to put **Localhost:8000/redoc**

The screenshot displays the Redoc API documentation interface. It features a sidebar on the left with a search bar and several API endpoints listed as green buttons:

- Index** (GET)
- About** (GET) (listed twice)

The main content area is divided into three sections:

- Index**: Shows a "Responses" section with a button for "200 Successful Response".
- About**: Shows a "Responses" section with a button for "200 Successful Response".
- /blog/{id}**: A detailed view of the endpoint. It shows a "Response samples" section for status 200, which includes "Content type application/json" and a "null" response body. It also shows another "Response samples" section for status 200, identical to the first. Below these, there are two buttons for status 200 and 422, and a "Content type application/json" section.

At the bottom left of the main content area, it says "API docs by Redocly".

QUERY PARAMETER

```
← C ⓘ 127.0.0.1:8000/test?limit=50&pub=true  
1 {  
2   "data": "50 published from the db"  
3 }
```

```
@app.get('/test')  
def test (limit, pub: bool):  
    if pub == True:  
        return {'data': f'{limit} published from the db'}  
    else:  
        return {'data': f'{limit} not existed from the db'}
```

```
@app.get('/test')  
def test (limit = 10, pub: bool = True):  
    if pub == True:  
        return {'data': f'{limit} published from the db'}  
    else:  
        return {'data': f'{limit} not existed from the db'}
```

For default value to be more pertinent when the user didn't enter the limit and the bool

QUERY PARAMETER

If it is not given by the user it will default to 10

```
def test (limit = 10, pub: bool = True, sort: Optional[str] = None):  
    from typing import Optional
```

To add optional value pour que l'utilisateur peut la entrer.

REQUEST BODY

- When you need to send data from a client (let's say, a browser) to your API, you send it as a request body.
- To declare a request body, you use **Pydantic**.

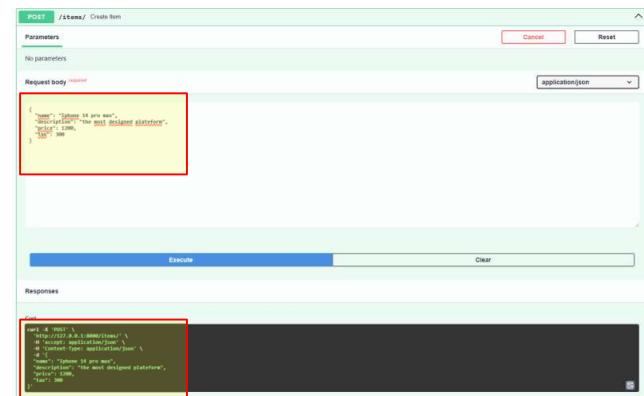
```
from pydantic import BaseModel
```

```
class Item(BaseModel):  
    name: str  
    description: str | None = None  
    price: float  
    tax: float | None = None
```



```
@app.post("/items/")  
async def create_item(item: Item):  
    return item
```

<http://127.0.0.1:8000/docs>



HOW TO DEBUG

```
if __name__ == "__main__":
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

To CHANGE THE PORT OF SERVER UVICORN

FastAPI doesn't require you to use a **SQL** (relational) database.

But you can use any relational database that you want.

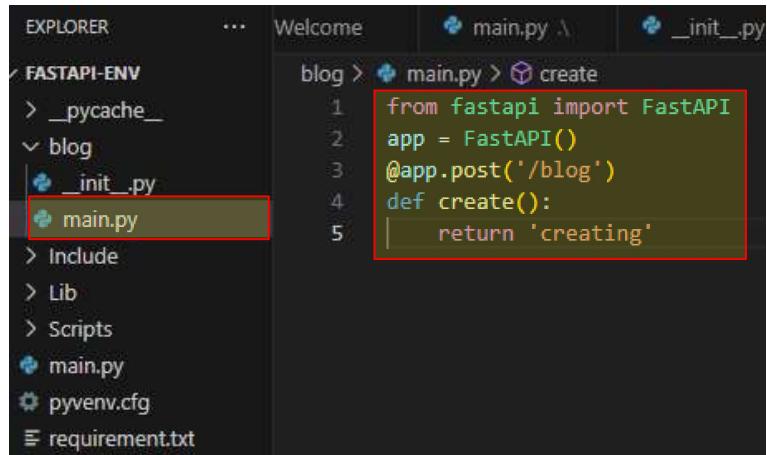
Here we'll see an example using **SQLAlchemy** [↪].

You can easily adapt it to any database supported by **SQLAlchemy**, like:

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server, etc.

DATA BASE AND CONNECTIONS

Create the database



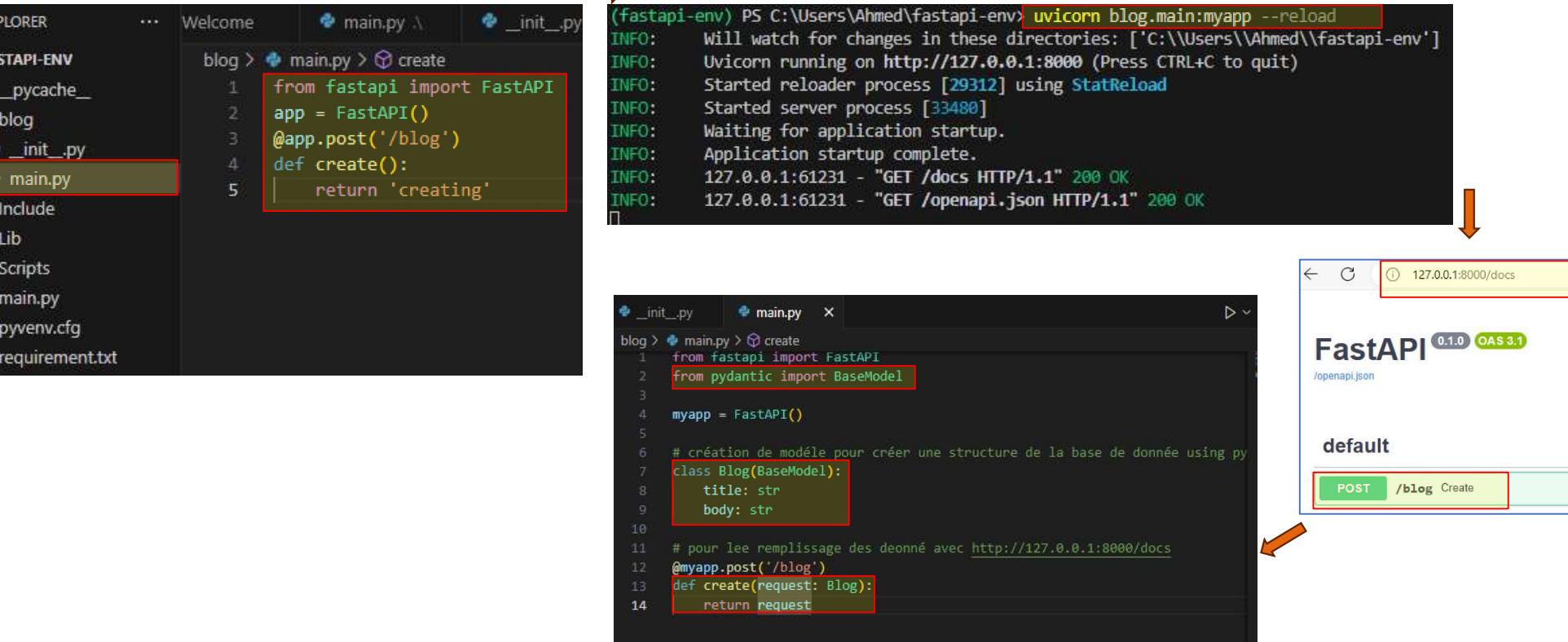
EXPLORER ... Welcome main.py __init__.py
FASTAPI-ENV
> _pycache_
blog
|__ __init__.py
└── main.py

blog > main.py > create

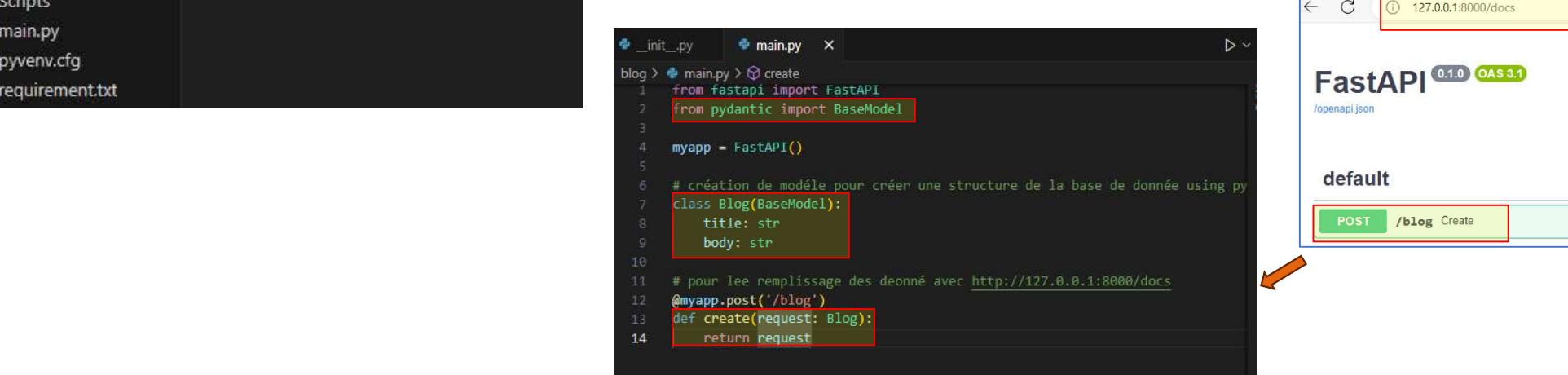
```
1 from fastapi import FastAPI
2 app = FastAPI()
3 @app.post('/blog')
4 def create():
5     return 'creating'
```

Include
Lib
Scripts
main.py
pyenv.cfg
requirement.txt

We need to put blog.main because the path
is in a folder called blog



```
(fastapi-env) PS C:\Users\Ahmed\fastapi-env> uvicorn blog.main:myapp --reload
INFO: Will watch for changes in these directories: ['C:\\\\Users\\\\Ahmed\\\\fastapi-env']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [29312] using StatReload
INFO: Started server process [33480]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:61231 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:61231 - "GET /openapi.json HTTP/1.1" 200 OK
```

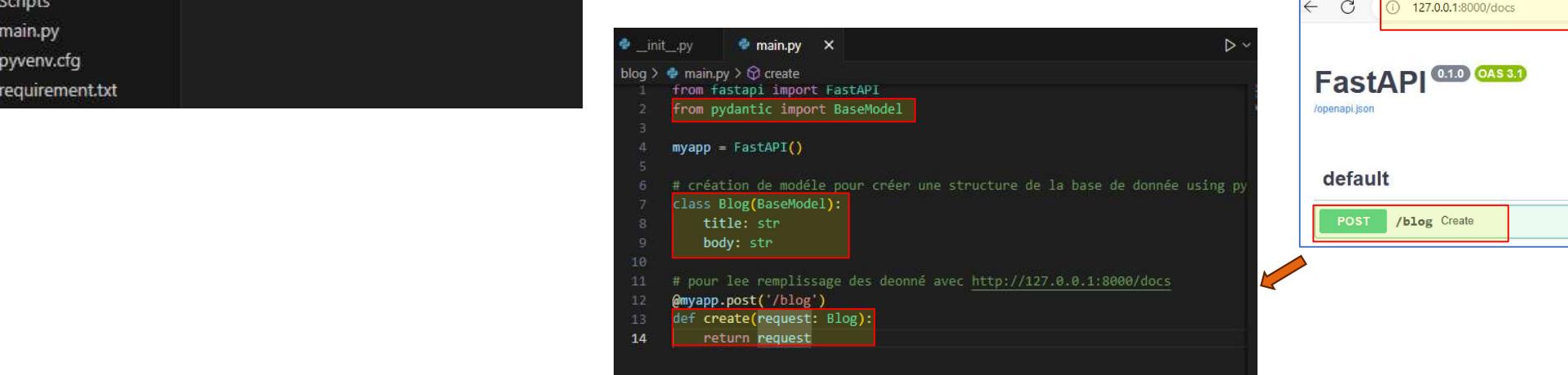


← ⌂ ⓘ 127.0.0.1:8000/docs

FastAPI 0.1.0 OAS 3.1
/openapi.json

default

POST /blog Create



```
__init__.py main.py X
blog > main.py > create
1 from fastapi import FastAPI
2 from pydantic import BaseModel
3
4 myapp = FastAPI()
5
6 # création de modèle pour créer une structure de la base de donnée using py
7 class Blog(BaseModel):
8     title: str
9     body: str
10
11 # pour le remplissage des données avec http://127.0.0.1:8000/docs
12 @myapp.post('/blog')
13 def create(request: Blog):
14     return request
```

DATA BASE AND CONNECTIONS

To organize the giveaways:

- 1- Creation of diagrams. (for the model)
- 2- Call the main page template

The screenshot shows a code editor interface with two tabs: `__init__.py` and `main.py`. The `main.py` tab is active, displaying the following code:

```
from fastapi import FastAPI
from . import schemas

myapp = FastAPI()

# pour le remplissage des données avec http://127.0.0.1:8000/docs
@app.post('/blog')
def create(request: schemas.Blog):
    return request
```

The `schemas.py` file in the Explorer is highlighted with a red box. The code in the `schemas.py` tab is also highlighted with a red box, indicating it is the source of the imported module. A yellow box highlights the `Main.py` tab.

EXPLORER

- FASTAPI-ENV
- > __pycache__
- > blog
- > __pycache__
- __init__.py
- main.py
- schemas.py**
- > Include
- > Lib
- > Scripts
- main.py
- pyvenv.cfg
- requirement.txt

... `__init__.py` `main.py` `schemas.py`

blog > `schemas.py` > ...

```
1 from pydantic import BaseModel
2
3
4 # création de modèle pour créer une structure de la base de données en utilisant pydantic
5 class Blog(BaseModel):
6     title: str
7     body: str
```

from fastapi import FastAPI
from . import schemas

myapp = FastAPI()

pour le remplissage des données avec <http://127.0.0.1:8000/docs>
@myapp.post('/blog')
def create(request: schemas.Blog):
 return request

Main.py

DATA BASE AND CONNECTIONS

- 1- **FastAPI works with any database** and any style of library to talk to the database.
- 2- A **common pattern is to use an "ORM": an "object-relational mapping" library**.
- 3- An ORM has tools to **convert ("map") between objects in code and database tables** ("relations").
- 4- With an ORM, you normally create a class that represents a table in a SQL database, each attribute of the class represents a column, with a name and a type.

1- Install sqlalchemy

```
(fastapi-env) PS C:\Users\Ahmed\fastapi-env> pip install sqlalchemy
```

2- create blog.db under blog folder

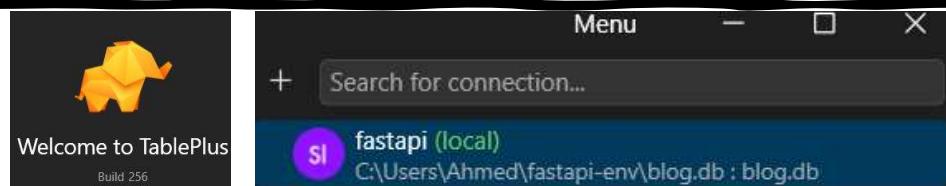
The screenshot shows a code editor interface with a dark theme. On the left, there's an Explorer sidebar listing files and folders: main.py, database.py, blog.db, .pycache_, __init__.py, database.py (highlighted with a red box), main.py, schemas.py, and requirement.txt. The main editor area displays a Python script named database.py:

```
blog > database.py > ...
1  from sqlalchemy import create_engine
2  from sqlalchemy.ext.declarative import declarative_base
3  from sqlalchemy.orm import sessionmaker
4
5  # to connect we use create_engine and fastapis uses connection connect_args as false
6  SQLALCHEMY_DATABASE_URL = "sqlite:///./blog.db"
7  engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
8
9  # create the session
10 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
11
12 # declare a mapping
13 Base = declarative_base()
```

3- create a database.py (where we're going to put the connections and map ORM)

MODEL & TABLE

- 1- use of table plus to vizualise database and connect to blog database.



- 2- create schema of the data base which called models. File models.py and

A screenshot of a code editor showing two files: 'models.py' and 'main.py'.
models.py:

```
from sqlalchemy import Column, Integer, String
from .database import Base

class Blog(Base):
    # le nom de tableau
    __tablename__ = 'blogs'
    # les colonnes de la base de donnée declarer
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String)
    body = Column(String)
```

main.py:

```
from fastapi import FastAPI
from . import schemas, models
from .database import engine

myapp = FastAPI()
models.Base.metadata.create_all(engine)

@myapp.post('/blog')
def create(request: schemas.Blog):
    return request
```

The code editor has several UI elements: an 'EXPLORER' sidebar on the left listing files like 'database.py', 'models.py' (highlighted with a red box), 'schemas.py', and 'main.py'; a central workspace with tabs for 'database.py', 'models.py' (highlighted with a red box), 'Untitled-1', and 'blog.db'; and a bottom toolbar with icons for file operations and a search bar. A yellow box highlights the 'results.py' tab in the bottom right corner.

STORE ELEMENT TO THE DATABASE WITH FASTAPI

- Next fastapi set will be used to add new contents. Main.py

```
from fastapi import FastAPI, Depends
from . import schemas, models
from .database import engine, SessionLocal
from sqlalchemy.orm import Session  # Database session use

app = FastAPI()

models.Base.metadata.create_all(engine)

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.post('/blog')
def create(request: schemas.Blog, db: Session = Depends(get_db)):
    new_blog = models.Blog(title=request.title, body=request.body)
    db.add(new_blog)
    db.commit()
    db.refresh(new_blog)
    return new_blog
```

Database session use

For declaring variable db as a database

Use and set new arguments or user variable in the database

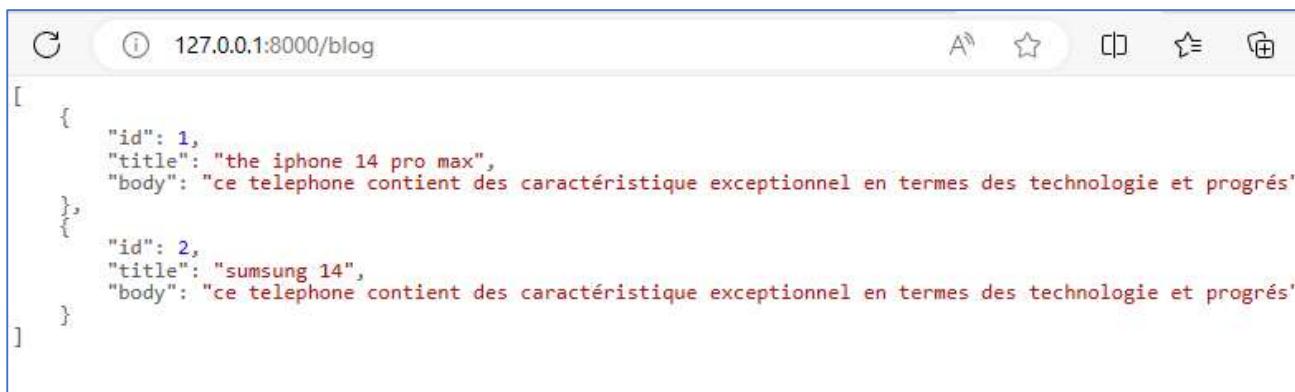
GET DATA FROM DATABASE USING FASTAPI

En main.py

```
@myapp.get('/blog')
def all(db: Session=Depends(get_db)):
    blogs = db.query(models.Blog).all()
    return blogs
```

To say that run a query to return all the data
in the db (all)

- To get all blogs using fastapi we add the next code.



```
[{"id": 1, "title": "the iphone 14 pro max", "body": "ce telephone contient des caract\u00e9ristique exceptionnel en termes des technologie et progr\u00e8s"}, {"id": 2, "title": "samsung 14", "body": "ce telephone contient des caract\u00e9ristique exceptionnel en termes des technologie et progr\u00e8s"}]
```

For more examples:

fastapi.tiangolo.com/tutorial/sql-databases/?h=sql#install-sqlalchemy

GET DATA FROM DATABASE USING FASTAPI

- To get some specific result using fastapi we add the next code.

```
@app.get('/blog/{id}')
def show(id, db: Session = Depends(get_db)):
    blog = db.query(models.Blog).filter(models.Blog.id == id).first()
    return blog
```



For the execution of database-specific commands to give according to identifiers

EXCEPTION & STATUS CODE

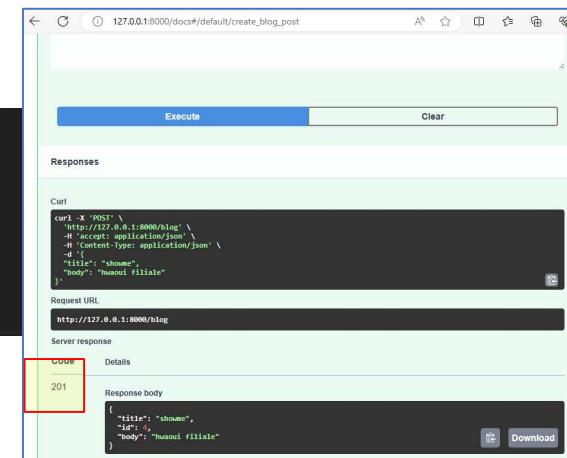
- When add a new data the response code based on documentation should be 201

documentation

```
from fastapi import FastAPI
app = FastAPI()
@app.post("/items/", status_code=201)
async def create_item(name: str):
    return {"name": name}
```

main.py

```
@myapp.post('/blog', status_code=201)
def create(request: schemas.Blog, db: Session = Depends(get_db)):
    new_blog = models.Blog(title=request.title, body=request.body)
    db.add(new_blog)
    db.commit()
    db.refresh(new_blog)
    return new_blog
```



To avoid looking for code 201 and avoid waste of time the solution is with Fastapi

```
from fastapi import FastAPI, Depends, status
@app.post('/blog', status_code=status.HTTP_201_CREATED)
def create(request: schemas.Blog, db: Session = Depends(get_db)):
    new_blog = models.Blog(title=request.title, body=request.body)
    db.add(new_blog)
    db.commit()
    db.refresh(new_blog)
    return new_blog
```

EXCEPTION & STATUS CODE

- When we request some value that doesn't exist in the data base the message error generated will as follows

Main.py

```
from fastapi import FastAPI, Depends, status, Response
@app.get('/blog/{id}', status_code=200)
def show(id, respance:Response, db : Session = Depends(get_db)):
    blog = db.query(models.Blog).filter(models.Blog.id == id).first()
    if not blog:
        respance.status_code = status.HTTP_404_NOT_FOUND
        return blog
```

- Adding this line the result will be more accurate:

```
return {'detail': f'the blog with the id : {id} not existed yet'}
```

http://127.0.0.1:8000/docs

Code	Details
404 <i>Undocumented</i>	Error: Not Found Response body <pre>{ "detail": "the blog with the id : 10 not existed yet" }</pre>



EXCEPTION & STATUS CODE

- When we request some value that doesn't exist in the data base the message error generated will as follows 2nd method

Main.py

```
from fastapi import FastAPI, Depends, status, Response, HTTPException

@app.get('/blog/{id}', status_code=200)
def show(id, respance:Response, db : Session = Depends(get_db)):
    blog = db.query(models.Blog).filter(models.Blog.id == id).first()
    if not blog:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'the blog with the id : {id} not existed yet')
    #respance.status_code = status.HTTP_404_NOT_FOUND
    #return {'detail': f'the blog with the id : {id} not existed yet'}
    return blog
```



DELETE A BLOG

Main.py

```
@myapp.delete('/blog/{id}', status_code=status.HTTP_204_NO_CONTENT)
def destroy(id, db: Session = Depends(get_db)):
    blog = db.query(models.Blog).filter(models.Blog.id == id).delete(synchronize_session=False)
    db.commit()

    return 'done'
```

http://127.0.0.1:8000/docs

default

The screenshot shows the FastAPI documentation interface for the 'default' endpoint. It lists four operations: GET /blog All, POST /blog Create, GET /blog/{id} Show, and DELETE /blog/{id} Destroy. The 'DELETE /blog/{id} Destroy' button is highlighted with a red border, indicating it is the current selection or the one being demonstrated.

Method	Path	Description
GET	/blog All	
POST	/blog Create	
GET	/blog/{id} Show	
DELETE	/blog/{id} Destroy	

UPDATE A BLOG (PUT)

Main.py

```
@myapp.put('/blog/{id}', status_code=status.HTTP_202_ACCEPTED)
def update(id, request: schemas.Blog, db: Session = Depends(get_db)):
    blog = db.query(models.Blog).filter(models.Blog.id == id)
    print(models.Blog.id)
    if not blog.first():
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'the id: {id} not found')
    blog.update({models.Blog.title: request.title, models.Blog.body: request.body})
    db.commit()
    return 'updated'
```

- The schema is necessary for the blog content format.

http://127.0.0.1:8000/docs

PUT /blog/{id} Update



RESPONSE MODEL: RESPONSE SCHEMA (PYNDANTIC MODEL)

The image shows a code editor interface with two files and a results section.

Schemas.py

```
models.py
schemas.py
> Include
> Lib
> Scripts
≡ blog.db
main.py
# uses for affichage bien determiner pour seulement title
class showblog(BaseModel):
    title: str
    class Config():
        orm_mode = True
```

main.py

```
# to get specific information from the database
@app.get('/blog/{id}', status_code=200, response_model=schemas.showblog)
def show(id, respance:Response, db : Session = Depends(get_db)):
    blog = db.query(models.Blog).filter(models.Blog.id == id).first()
    if not blog:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'the blog with the id : {id} not existed yet')
        #respance.status_code = status.HTTP_404_NOT_FOUND
        #return {'detail': f'the blog with the id : {id} not existed yet'}
    return blog
```

result

200	Response body
	{ "title": "the iphone 14 pro max" }

CREATE USER

The diagram illustrates the relationship between the `schemas.py` and `Main.py` files. On the left, the `schemas.py` file is shown with its code structure. An orange arrow points from the `User` class definition in `schemas.py` to the `@myapp.post('/user')` endpoint in `Main.py`. Another orange arrow points from the `POST /user Create User` endpoint back to the `create_user` function in `Main.py`.

```
schemas.py
16  class User(BaseModel):
17      name: str
18      email: str
19      password: str
```

```
Main.py
@myapp.post('/user')
def create_user(request: schemas.User):
    return request
```

```
POST /user Create User
```

- Création de la table user

The diagram shows the creation of the `users` table and its data. On the left, the `models.py` file is displayed, containing the definition of the `User` class. An orange arrow points from the `User` class definition in `models.py` to the `create_user` function in `Main.py`. The `create_user` function in `Main.py` is shown creating a new `User` object with the name "Ahmed", email "ahmed@whize.fr", and password "Hello Whize". This function then adds the user to the database session, commits the changes, and refreshes the new user object. On the right, a screenshot of a database interface shows two tables: `blogs` and `users`. The `users` table has one row with the id 1, name "Ahmed", email "ahmed@whize.fr", and password "Hello Whize".

```
models.py
11  class User(Base):
12      __tablename__ = 'users'
13
14      id = Column(Integer, primary_key=True, index=True)
15      name = Column(String)
16      email = Column(String)
17      password = Column(String)
```

```
Main.py
@myapp.post('/user')
def create_user(request: schemas.User, db : Session = Depends(get_db)):
    new_user = models.User(name=request.name, email=request.email, password=request.password)
    db.add(new_user)
    db.commit()
    db.refresh(new_user)
    return new_user
```

id	name	email	password
1	Ahmed	ahmed@whize.fr	Hello Whize

PASSWORD ENCRYPTION

- Install passlib library

```
(fastapi-env) PS C:\Users\Ahmed\fastapi-env> pip install "passlib[bcrypt]"
```

Main.py

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

@app.post('/user')
def create_user(request: schemas.User, db : Session = Depends(get_db)):
    hashedpassword= pwd_context.hash(request.password)
    new_user = models.User(name=request.name, email=request.email, password=hashedpassword)
    db.add(new_user)
    db.commit()
    db.refresh(new_user)
    return new_user
```

http://127.0.0.1:8000/docs#

```
{
    "email": "Guizani",
    "id": 2,
    "name": "Ahmed",
    "password": "$2b$12$Y0uJoBtRN1wUTUa.jn15t0p.zYPfcsgPnRVN4nfCiAD.0Wa2xNp4a"
}
```

sqlite

2 Ahmed

Guizani

\$2b\$12\$Y0uJoBtRN1wUTUa.jn15t0...

USER ID GET

Main.py

```
@myapp.get('/user/{id}', status_code=200, response_model=schemas.showUser)
def show(id, db : Session = Depends(get_db)):
    user = db.query(models.User).filter(models.User.id == id).first()
    if not user:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'the user with the id : {id} not existed')
    #responce.status_code = status.HTTP_404_NOT_FOUND
    #return {'detail': f'the blog with the id : {id} not existed yet'}
    return user
```

Schemas.py
Pour afficher que nom et
email sans mdp

```
class showUser(BaseModel):
    name: str
    email: str
```

http://127.0.0.1:8000/docs

200

Response body

```
{
    "name": "daniel",
    "email": "daniel@whize.fr"
}
```

USING DOCS TAGS

Main.py



```
@myapp.put('/blog/{id}', status_code=status.HTTP_202_ACCEPTED, tags=['blogs'])
```

```
@myapp.get('/user/{id}', status_code=200, response_model=schemas.showUser, tags=['users'])
```

RELATIONSHIP API (FOREIGN KEY)

Every blog should
belong to user

models.py

```
from sqlalchemy import Column, Integer, String, ForeignKey
from .database import Base
from sqlalchemy.orm import relationship

class Blog(Base):
    # le nom de tableau
    __tablename__ = 'blogs'
    # les colonnes de la base de donnée déclarer
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String)
    body = Column(String)
    user_id = Column(Integer, ForeignKey("users.id"))
    creator = relationship("User", back_populates="blogs")

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String)
    email = Column(String)
    password = Column(String)
    blogs = relationship('Blog', back_populates="creator")
```

Needs to delete
table already created

main.py

```
@myapp.post('/blog', status_code=status.HTTP_201_CREATED, tags=['blogs'])
def create(request: schemas.Blog, db: Session = Depends(get_db)):
    new_blog = models.Blog(title=request.title, body=request.body, user_id=1)
    db.add(new_blog)
    db.commit()
    db.refresh(new_blog)
```

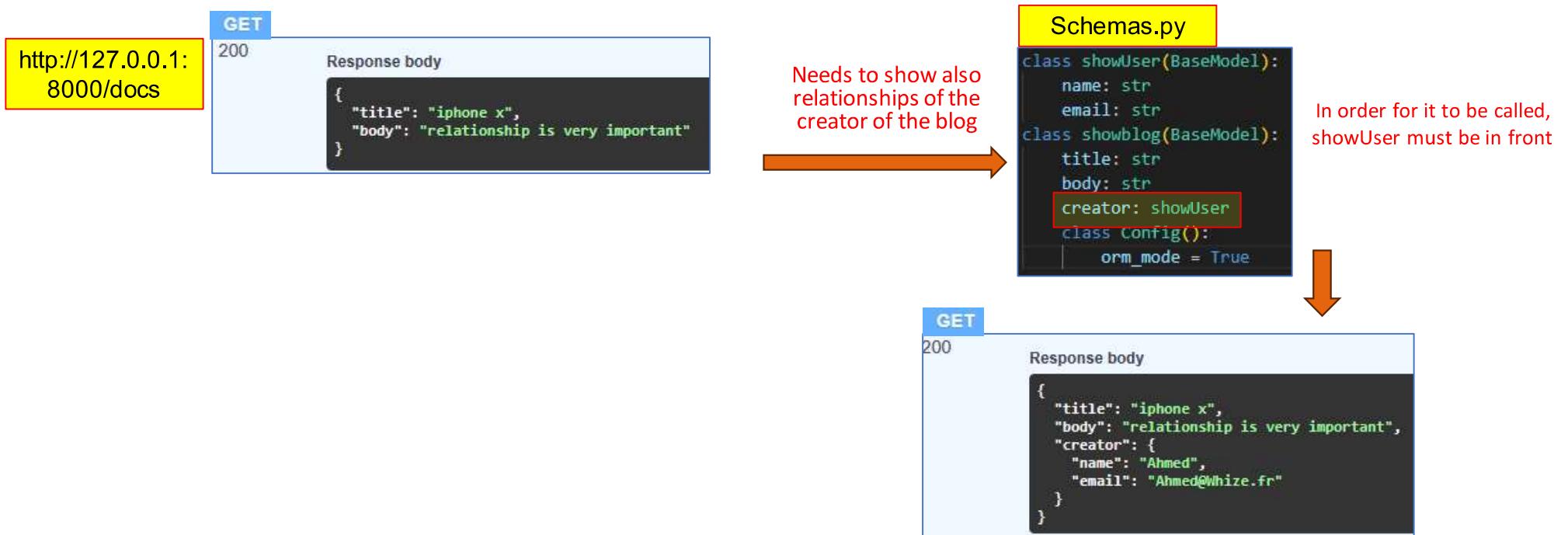
http://127.0.0.1:8000/docs

201

Response body

```
{
    "body": "relationship is very important",
    "title": "iphone x",
    "id": 1,
    "user_id": 1
}
```

RELATIONSHIP API (FOREIGN KEY)



RELATIONSHIP API (FOREIGN KEY)

Needs to show also the blogs for each user

Schemas.py

```
class showUser(BaseModel):
    name: str
    email: str
    blogs : list[Blog] = []
class Config():
    orm_mode = True
```

In order for the user display to be complete, need to call list

models.py

```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String)
    email = Column(String)
    password = Column(String)
    blogs = relationship('Blog', back_populates="creator")
```

http://127.0.0.1:
8000/docs

GET

200

Response body

```
{
  "name": "Ahmed",
  "email": "Ahmed@hize.fr",
  "blogs": [
    {
      "title": "iphone x",
      "body": "relationship is very important"
    },
    {
      "title": "iphone 13 pro max",
      "body": "i wrote for this phone because of their composition and cameras"
    }
]
```

API ROUTER/ FOR BIGGER APPLICATIONS REFACTORING



API ROUTER OPERATOR

Blog.py

```
@router.get('/blog', tags=['blogs'])
def all(db: Session=Depends(get_db)):
    blogs = db.query(models.Blog).all()
    return blogs
```

To avoid redendance
of /blod or tags

Blog.py

```
router = APIRouter(
    prefix='/blog',
    tags=['BLogs']
)
```

```
@router.get('/')
```

Same for
user.py

BLOG AND USER REPOSITORIES

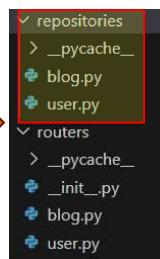
Routers must be very well organized to differentiate that this directory is only for .py routing

```
from fastapi import APIRouter, Depends, HTTPException, status
from .. import models, schemas
from ..database import get_db
from sqlalchemy.orm import Session
from passlib.context import CryptContext

router = APIRouter(
    prefix='/user',
    tags=['users']
)
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

@router.post('/')
def create_user(request: schemas.User, db : Session = Depends(get_db)):
    hashedpassword= pwd_context.hash(request.password)
    new_user = models.User(name=request.name, email=request.email, password=hashedpassword)
    db.add(new_user)
    db.commit()
    db.refresh(new_user)
    return new_user

@router.get('/{id}', status_code=200, response_model=schemas.showUser)
def show(id, db : Session = Depends(get_db)):
    user = db.query(models.User).filter(models.User.id == id).first()
    if not user:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'the user with the id : {id} not existed')
    return user
```



```
from fastapi import APIRouter, Depends, HTTPException, status
from .. import models, schemas
from ..database import get_db
from sqlalchemy.orm import Session
from passlib.context import CryptContext
from ..repositories import user

router = APIRouter(
    prefix='/user',
    tags=['users']
)
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

@router.post('/')
def create_user(request: schemas.User, db : Session = Depends(get_db)):
    return user.create_user(request, db)

@router.get('/{id}', status_code=200, response_model=schemas.showUser)
def show(id: int, db : Session = Depends(get_db)):
    return user.show(id, db)
```

routers/user.py

Same for blog

```
from fastapi import HTTPException, status
from .. import models, schemas
from sqlalchemy.orm import Session
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def create_user(request: schemas.User, db : Session):
    hashedpassword= pwd_context.hash(request.password)
    new_user = models.User(name=request.name, email=request.email, password=hashedpassword)
    db.add(new_user)
    db.commit()
    db.refresh(new_user)
    return new_user
```

Repositories/user.py

LOGIN AND VERIFY PASSWORD

```
✓ routers
  > __pycache__
  init.py
  authentication.py
  blog.py
  user.py
```

1- Creating an AUTH file in Routiers Repo

```
authentication.py
```

```
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from .. import schemas, database, models, Hashing

router = APIRouter(
    prefix='/login',
    tags=['Login'])

@router.post('/')
def login(request: schemas.Login, db : Session = Depends(database.get_db)):
    user = db.query(models.User).filter(models.User.email == request.username).first()
    if not user:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'invalid credential')
    # verifying the password
    if not Hashing.Hash.verify(user.password, request.password):
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'Incorrect password')
    # generate the jwt token and return it
    return user
```

2- Code creation for
the login file

```
Hashing.py (inside blog)
```

```
class Hash():
    def bcrypt(password: str):
        return pwd_context.hash(password)
    def verify(hashedpwd, plainpwd):
        return pwd_context.verify(plainpwd, hashedpwd)
```

```
Main.py
```

```
from .routers import authentication
myapp.include_router(authentication.router)
```



LOGIN AND VERIFY PASSWORD

```
{  
    "username": "taha@whize.fr",  
    "password": "123"  
}
```

404
Undocumented

Error: Not Found

Response body

```
{  
    "detail": "Incorrect password"  
}
```



```
{  
    "username": "hhhhhhhhhh",  
    "password": "taha"  
}
```

Code Details

404
Undocumented

Error: Not Found

Response body

```
{  
    "detail": "invalid credential"  
}
```



```
{  
    "username": "taha@whize.fr",  
    "password": "taha"  
}
```

Code Details

200

Response body

```
{  
    "password": "$2b$12$kGQm75ZiaHgtk0yCiiDIY..BK1OTJumP0iEyd9RrovmiYbe9beFHS",  
    "id": 5,  
    "email": "taha@whize.fr",  
    "name": "taha"  
}
```



Le code

```
@router.post('/')
```

```
def login(request: schemas.Login, db : Session = Depends(database.get_db)):  
    user = db.query(models.User).filter(models.User.email == request.username).first()  
    if not user:  
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,  
                            detail=f'invalid credential')  
  
    # verifying the password  
    if not Hashing.Hash.verify(user.password, request.password):  
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,  
                            detail=f' Incorrect password')  
  
    # generate the jwt token and return it  
    return user
```

JWT ACCESS TOKEN

(FASTAPI OAUTH2 WITH PASSWORD (AND HASHING), BEARER WITH JWT TOKENS)

fastapi.tiangolo.com/tutorial/security/oauth2-jwt/?h=oauth

1- install a librarie
python-jose

```
(fastapi-env) PS C:\Users\Ahmed\fastapi-env> pip install python-jose
```

2- create file token
(for JWT)



```
class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    username: str | None = None
```



3- add
schemas basemodel

```
authentication.py 1      token.py x      schemas.py      Hashing.py      main.py
blog > token.py > create_access_token
from datetime import datetime, timedelta, timezone
from jose import JWTError, jwt

SECRET_KEY = "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
    expire = datetime.now(timezone.utc) + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

4- add the generate
jwt token



```
# generate the jwt token and return it
access_token = token.create_access_token(data={"sub": user.email})
return {"access_token": access_token, "token_type": "bearer"}
```

Login
POST /login/ Login



```
{
    "username": "taha@whize.fr",
    "password": "taha"
}
```



Code	Details
200	<p>Response body</p> <pre>{"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0YWhhQHdoaXplLmZyIiwizXhwIJoxNzA2NTA1MjYzFQ.tGcx2FeZLj-Ay7PCJki45_tBlnRLk48nQ8rAOGzfUgg", "token_type": "bearer"</pre>

ROUTES BEHIND AUTHENTICATION

fastapi.tiangolo.com/tutorial/security/oauth2-jwt/?h=oauth
FastAPI - A python framework | Full Course (youtube.com)
SQL (Relational) Databases - FastAPI (tiangolo.com)

1- create file oauth2.py
(we will use JWT)

```
from fastapi import Depends, HTTPException, status
from typing import Annotated
from fastapi.security import OAuth2PasswordBearer
from . import token

# ROUTE OR URL FROM WHERE FASTAPI WILL FETCH THE TOKEN
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
def get_current_user(token: Annotated[str, Depends(oauth2_scheme)]):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    return token.verify_token(token, credentials_exception)
```

2- add to token.py
verify_token
function

```
def verify_token(token: str, credentials_exception):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        email: str = payload.get("sub")
        if email is None:
            raise credentials_exception
        token_data = schemas.TokenData(email=email)
    except JWTError:
        raise credentials_exception
```

3- add to the blog
verification of token

```
@router.get('/')
def all(db: Session=Depends(get_db), get_current_user: schemas.User = Depends(ouath2.get_current_user)):
    return blog.all(db)
```

4- modification of the
schemas

```
class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    email: str | None = None
```

5- result

