

# Guia Completo de Desenvolvimento de API com FastAPI e MongoDB

## 1. Introdução e Objetivos

Este guia detalha o processo de construção de uma API RESTful utilizando FastAPI, um framework web moderno e de alta performance para Python, e MongoDB como banco de dados NoSQL. A aplicação será containerizada com Docker e Docker Compose para facilitar o desenvolvimento e a implantação. O objetivo é fornecer um passo a passo completo, desde a configuração do ambiente até a implementação de funcionalidades CRUD (Create, Read, Update, Delete) com filtros avançados e paginação, além de incluir as melhores práticas para testes e resolução de problemas.

## 2. Pré-requisitos

Para seguir este guia, você precisará ter os seguintes softwares instalados em sua máquina:

- **Python 3.9+:** A linguagem de programação principal.
- **Docker e Docker Compose:** Para containerização da aplicação e do banco de dados. Certifique-se de que estão instalados e funcionando corretamente. No Windows, é recomendado o uso do Docker Desktop com WSL2 habilitado.
- **Um editor de código:** Como VS Code, PyCharm ou similar.
- **Ferramenta de teste de API:** Insomnia ou Postman (opcional, mas recomendado para testes manuais).
- **MongoDB Compass:** Para visualizar e gerenciar os dados no MongoDB (opcional).

## 3. Estrutura do Projeto

A estrutura de pastas do projeto será organizada da seguinte forma:

fastapi-mongo/

├─ .env

├─ .env.example

├─ requirements.txt

├─ Dockerfile

├─ docker-compose.yml

└─ app/

├─ \_\_init\_\_.py

├─ main.py

├─ config.py

├─ db.py

├─ models.py

└─ routers/

├─ \_\_init\_\_.py

└─ users.py

Crie as pastas `app/` e `app/routers/` e os arquivos `__init__.py` vazios dentro delas para que o Python reconheça `app` e `app/routers` como pacotes.

## 4. Configuração do Ambiente

### 4.1. `requirements.txt`

Este arquivo lista todas as dependências Python necessárias para o projeto. Certifique-se de que as versões estejam fixadas para evitar problemas de compatibilidade, especialmente entre `motor` e `pymongo`.

`fastapi==0.115.0`

`uvicorn[standard]==0.30.6`

`motor==3.5.1`

`pymongo==4.8.0`

`pydantic==2.9.2`

`pydantic-settings==2.6.0`

```
python-dotenv==1.0.1
```

```
email-validator==2.1.1
```

## 4.2. Dockerfile

O **Dockerfile** define como a imagem Docker da sua aplicação FastAPI será construída.

```
FROM python:3.12-slim
```

```
ENV PYTHONDONTWRITEBYTECODE=1 \
```

```
    PYTHONUNBUFFERED=1 \
```

```
    PIP_NO_CACHE_DIR=1
```

```
WORKDIR /app
```

```
RUN python -m pip install --upgrade pip
```

```
COPY requirements.txt .
```

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

```
COPY app ./app
```

```
EXPOSE 8000
```

```
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

## 4.3. docker-compose.yml

Este arquivo orquestra os serviços Docker (MongoDB e a API FastAPI), definindo como eles se comunicam e suas configurações.

```
version: "3.9"
```

```
services:
```

```
    mongo:
```

```
        image: mongo:7
```

container\_name: mongo

restart: unless-stopped

ports:

- "27017:27017"

volumes:

- mongo\_data:/data/db

healthcheck:

test: ["CMD-SHELL", "mongosh --quiet --eval 'db.runCommand({ ping: 1 })' || exit 1"]

interval: 5s

timeout: 3s

retries: 20

api:

build: .

container\_name: fastapi-mongo

restart: unless-stopped

env\_file: .env

environment:

# Aponta para o serviço 'mongo' dentro da rede do Compose:

MONGO\_URI: mongodb://mongo:27017

volumes:

- ./app:/app/app:rw

command: uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload

ports:

- "8000:8000"

depends\_on:

mongo:

condition: service\_healthy

volumes:

mongo\_data:

#### 4.4. `.env.example` e `.env`

O arquivo `.env.example` serve como modelo para as variáveis de ambiente. Copie seu conteúdo para um arquivo `.env` na raiz do projeto e ajuste conforme necessário. O `docker-compose.yml` sobrescreve `MONGO_URI` para `mongodb://mongo:27017` dentro da rede do Docker Compose, garantindo que a API se conecte ao serviço `mongo`.

##### `.env.example`

`MONGO_URI=mongodb://localhost:27017`

`MONGO_DB=faculdade`

`APP_NAME=Aula 04 - FastAPI + MongoDB`

##### `.env`

`MONGO_URI=mongodb://localhost:27017`

`MONGO_DB=faculdade`

`APP_NAME=Aula 04 - FastAPI + MongoDB`

## 5. Implementação da API

### 5.1. `app/config.py`

Define as configurações da aplicação, carregando variáveis de ambiente com `pydantic-settings`.

```
from pydantic_settings import BaseSettings

from pydantic import Field

class Settings(BaseSettings):

    MONGO_URI: str = Field("mongodb://mongo:27017", alias="MONGO_URI")

    MONGO_DB: str = Field("faculdade", alias="MONGO_DB")

    APP_NAME: str = Field("FastAPI + Mongo", alias="APP_NAME")

    class Config:

        env_file = ".env"

        extra = "ignore"

settings = Settings()
```

### 5.2. `app/db.py`

Gerencia a conexão com o MongoDB, incluindo lógica de retry para garantir que a API espere o banco de dados estar pronto, especialmente em ambientes Docker Compose.

```
import asyncio

from motor.motor_asyncio import AsyncIOMotorClient

from .config import settings

client: AsyncIOMotorClient | None = None

db = None
```

```

async def connect_to_mongo(max_retries: int = 30, delay: float = 1.0):
    """
    Tenta conectar com retries (útil quando o Mongo ainda está subindo no docker-compose).
    """

    global client, db

    for attempt in range(1, max_retries + 1):

        try:

            client = AsyncIOMotorClient(settings.MONGO_URI, serverSelectionTimeoutMS=2000)

            # força um ping

            await client.admin.command("ping")

            db = client[settings.MONGO_DB]

            # índices úteis (email único na coleção users)

            await db.users.create_index("email", unique=True)

            return

        except Exception:

            if attempt == max_retries:

                raise

            await asyncio.sleep(delay)

    async def close_mongo_connection():

        global client

        if client:

            client.close()

```

### 5.3. `app/models.py`

Define os modelos de dados para a API usando Pydantic, incluindo a serialização correta de `ObjectId` do MongoDB.

```
from typing import Optional, Annotated

from pydantic import BaseModel, EmailStr, Field, field_serializer

from pydantic.functional_validators import BeforeValidator

from bson import ObjectId

def _validate_object_id(v):

    if isinstance(v, ObjectId):

        return v

    if isinstance(v, str) and ObjectId.is_valid(v):

        return ObjectId(v)

    raise ValueError("Invalid ObjectId")

PyObjectId = Annotated[ObjectId, BeforeValidator(_validate_object_id)]

class UserIn(BaseModel):

    name: str = Field(..., min_length=2, max_length=80)

    email: EmailStr

    age: Optional[int] = Field(None, ge=0, le=130)

    is_active: bool = True

class UserOut(UserIn):

    id: PyObjectId = Field(alias="_id")

    # Garante que o ObjectId sairá como string no JSON
```



```
@field_serializer("id")
```

```
def dump_object_id(self, v: ObjectId, _info):
```

```
    return str(v)
```

## 5.4. `app/routers/users.py`

Contém as rotas da API para a entidade `User`, implementando operações CRUD completas com filtros avançados (regex, gte/lte) e paginação.

```
from typing import List, Optional
```

```
from fastapi import APIRouter, HTTPException, Query, status
```

```
from bson import ObjectId
```

```
from pymongo import ReturnDocument
```

```
from ..db import db
```

```
from ..models import UserIn, UserOut
```

```
router = APIRouter(prefix="/users", tags=["users"])
```

```
def to_user_out(doc) -> UserOut:
```

```
    return UserOut.model_validate(doc)
```

```
@router.get("/", response_model=List[UserOut])
```

```
async def list_users(
```

```
    q: Optional[str] = Query(None, description="Busca por nome (regex, case-insensitive)"),
```

```
    min_age: Optional[int] = Query(None, ge=0),
```

```
    max_age: Optional[int] = Query(None, ge=0),
```

```
    is_active: Optional[bool] = Query(None),
```

```
    page: int = Query(1, ge=1),
```

```

limit: int = Query(10, ge=1, le=100),
):

filters: dict = {}

if q:
    filters["name"] = {"$regex": q, "$options": "i"}

if is_active is not None:
    filters["is_active"] = is_active

if min_age is not None or max_age is not None:
    age_filter = {}

    if min_age is not None:
        age_filter["$gte"] = min_age

    if max_age is not None:
        age_filter["$lte"] = max_age

    filters["age"] = age_filter

skip = (page - 1) * limit

cursor = db.users.find(filters).skip(skip).limit(limit).sort("name", 1)

docs = await cursor.to_list(length=limit)

return [to_user_out(d) for d in docs]

@router.get("/{user_id}", response_model=UserOut)

async def get_user(user_id: str):

    if not ObjectId.is_valid(user_id):

```

```

        raise HTTPException(status_code=400, detail="Invalid user id")

    doc = await db.users.find_one({"_id": ObjectId(user_id)})

    if not doc:

        raise HTTPException(status_code=404, detail="User not found")

    return to_user_out(doc)

@router.post("/", response_model=UserOut, status_code=status.HTTP_201_CREATED)
async def create_user(payload: UserIn):

    try:

        result = await db.users.insert_one(payload.model_dump())

        doc = await db.users.find_one({"_id": result.inserted_id})

        return to_user_out(doc)

    except Exception as e:

        if "E11000" in str(e):

            raise HTTPException(status_code=409, detail="Email already exists")

        raise

@router.put("/{user_id}", response_model=UserOut)
async def update_user(user_id: str, payload: UserIn):

    if not ObjectId.is_valid(user_id):

        raise HTTPException(status_code=400, detail="Invalid user id")

    res = await db.users.find_one_and_update(

        {"_id": ObjectId(user_id)},

        {"$set": payload.model_dump()},

```

```

        return_document=ReturnDocument.AFTER,
    )

    if not res:

        raise HTTPException(status_code=404, detail="User not found")

    return to_user_out(res)

@router.delete("/{user_id}", status_code=status.HTTP_204_NO_CONTENT)
async def delete_user(user_id: str):

    if not ObjectId.is_valid(user_id):

        raise HTTPException(status_code=400, detail="Invalid user id")

    res = await db.users.delete_one({"_id": ObjectId(user_id)})

    if res.deleted_count == 0:

        raise HTTPException(status_code=404, detail="User not found")

    return

```

## 5.5. `app/main.py`

O arquivo principal da aplicação FastAPI, onde a API é inicializada, as rotas são incluídas e o ciclo de vida da aplicação é gerenciado (conexão e desconexão com o MongoDB).

```

from fastapi import FastAPI

from contextlib import asynccontextmanager

from .config import settings

from .db import connect_to_mongo, close_mongo_connection

from .routers import users

@asynccontextmanager

```

```
async def lifespan(app: FastAPI):

    await connect_to_mongo()

    yield

    await close_mongo_connection()

app = FastAPI(title=settings.APP_NAME, lifespan=lifespan)

app.include_router(users.router)

@app.get("/")

async def root():

    return {"message": "API ok, vá em /docs para testar"}
```

## 6. Como Rodar a Aplicação

### 6.1. Pré-requisitos

Certifique-se de ter o Docker e Docker Compose instalados e funcionando.

```
docker --version
```

```
docker compose version
```

### 6.2. Baixar e Configurar o Projeto

1. Crie a estrutura de pastas conforme descrito na seção 3.
2. Crie os arquivos `__init__.py` vazios em `app/` e `app/routers/`.
3. Preencha os arquivos com o código fornecido nas seções anteriores.
4. Copie o conteúdo de `.env.example` para um novo arquivo chamado `.env` na raiz do projeto.

### 6.3. Subir os Serviços (Mongo + API)

No terminal, na raiz do projeto `fastapi-mongo/`, execute:

```
docker compose up --build
```

Este comando irá construir as imagens Docker (se necessário), criar e iniciar os contêineres do MongoDB e da API. Aguarde até que o MongoDB esteja saudável e a API inicie. Você pode verificar o status com:

```
docker compose ps
```

```
docker compose logs -f api
```

## 6.4. Testar a API

Após a API estar rodando, você pode testá-la de diversas formas:

Via Navegador (Swagger UI)

Abra seu navegador e acesse: <http://localhost:8000/docs>

Você verá a documentação interativa da API, onde pode executar os endpoints diretamente.

Via `curl` (Terminal)

### **Criar usuário (POST /users)**

```
curl -X POST http://localhost:8000/users \
```

```
-H "Content-Type: application/json" \
```

```
-d '{"name":"João","email":"joao@email.com","age":22}'
```

### **Listar usuários (GET /users)**

- Todos os usuários:

```
curl "http://localhost:8000/users"
```

- Busca por nome (regex, case-insensitive):

```
curl "http://localhost:8000/users?q=jo"
```

- Faixa de idade:

```
curl "http://localhost:8000/users?min_age=18&max_age=30"
```

- Paginação:

```
curl "http://localhost:8000/users?page=2&limit=5"
```

### Buscar usuário por ID (GET /users/{id})

Substitua `ID_AQUI` pelo `_id` retornado na criação ou listagem de usuários.

```
curl http://localhost:8000/users/ID_AQUI
```

### Atualizar usuário (PUT /users/{id})

Substitua `ID_AQUI` pelo `_id` do usuário a ser atualizado.

```
curl -X PUT http://localhost:8000/users/ID_AQUI \
-H "Content-Type: application/json" \
-d '{"name":"João Silva","email":"joao@ex.com","age":23,"is_active":false}'
```

### Excluir usuário (DELETE /users/{id})

Substitua `ID_AQUI` pelo `_id` do usuário a ser excluído.

```
curl -X DELETE http://localhost:8000/users/ID_AQUI
```

## 6.5. Visualizar Dados no MongoDB Compass

Abra o MongoDB Compass e conecte-se a `mongodb://localhost:27017`. Você poderá navegar até o banco de dados `faculdade` e ver a coleção `users` (que será criada automaticamente no primeiro `insert`).

## 6.6. Desenvolvendo com Live-Reload

O `docker-compose.yml` está configurado para montar o volume `./app:/app/app`. Isso significa que qualquer alteração nos arquivos Python dentro da pasta `app/` no seu sistema local será refletida automaticamente no contêiner da API, que recarregará a aplicação devido ao `--reload` no comando `uvicorn`.

## 6.7. Parar e Limpar

Para parar os serviços:

```
docker compose down
```

Para parar os serviços e remover o volume de dados do MongoDB (o que apagará todos os dados persistidos):

```
docker compose down -v
```

## 7. Troubleshooting e Correções Comuns

Esta seção aborda problemas comuns que podem surgir durante o desenvolvimento e como resolvê-los.

### 7.1. `ERR_EMPTY_RESPONSE` ou API não inicia

Este erro geralmente indica que o contêiner da API não está de pé ou não está ouvindo na porta esperada. Verifique os logs da API com `docker compose logs -f api`.

#### Causas e Soluções:

- **Contêiner caiu na inicialização:** Verifique os logs para mensagens de erro específicas.
- **`app/` não é um pacote Python:** Crie arquivos vazios `app/__init__.py` e `app/routers/__init__.py`.
- **Incompatibilidade `motor` e `pymongo`:** Fixe as versões no `requirements.txt` para `motor==3.5.1` e `pymongo==4.8.0`.
- **`email-validator` ausente:** Adicione `email-validator==2.1.1` ao `requirements.txt` ou instale `pydantic[email]`.
- **Variáveis de ambiente ausentes/incorretas:** Certifique-se de que o arquivo `.env` está presente e configurado corretamente. O `app/config.py` já inclui defaults seguros para evitar falhas.

#### Passos para aplicar correções e reconstruir:

```
docker compose down
```

```
docker compose build --no-cache
```

```
docker compose up
```

### 7.2. Portas Ocupadas

Se as portas 8000 (API) ou 27017 (MongoDB) já estiverem em uso em sua máquina, você pode alterá-las no `docker-compose.yml`:

ports:

```
- "8001:8000" # API
```



- "27018:27017" # Mongo (opcional)

Após a alteração, acesse a API em <http://localhost:8001/docs>.

### 7.3. Erro de E-mail Duplicado (E11000)

O MongoDB está configurado para ter um índice único no campo `email` da coleção `users`. Se você tentar criar um usuário com um e-mail já existente, a API retornará um erro `409 Conflict` com a mensagem "Email already exists". Isso é um comportamento esperado e uma boa prática para garantir a integridade dos dados.

### 7.4. Problemas de Montagem de Volume (Windows/Antivírus)

Em sistemas Windows, especialmente com antivírus, pode haver problemas de permissão ao montar volumes Docker. Tente rodar o terminal como administrador ou ajuste as permissões da pasta do projeto.

## 8. Considerações Finais e Treinos Posteriores

- **Relacionamentos:** Embora este guia use uma abordagem simples, explore a diferença entre documentos embutidos (`embedded documents`) e documentos referenciados (`referenced documents`) no MongoDB. Como atividade bônus, pode-se criar uma coleção `orders` com um `user_id` (referenciado).
- **Tratamento de Erros:** O erro `E11000` (duplicidade) é tratado na API, retornando um `HTTP 409 Conflict`, e discuta a importância de tratar erros de forma clara e informativa.
- **Filtros Avançados:** Reforço que operadores como `$gte`, `$lte`, `$in` e `$regex` são funcionalidades poderosas do MongoDB que podem ser aplicadas diretamente nos filtros da API, conectando com conceitos de consultas avançadas.
- **Documentação:** A documentação automática do FastAPI (Swagger UI) é uma ferramenta valiosa para testar e entender a API. Incentive o uso e a exploração dessa interface.

Este guia fornece uma base sólida para o desenvolvimento de APIs modernas e escaláveis com FastAPI e MongoDB, utilizando Docker para um ambiente de desenvolvimento consistente. Com ele, você tem todas as ferramentas para construir e testar sua aplicação de forma eficiente. Enjoy the process!