To Use FastAPI With Our Project:

* Run Command: ***pip install fastapi uvicorn[standard]***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Start Using FastAPI We Import It Then Use Decorators:

from fastapi import FastAPI;

app = FastAPI();

@app.get("/")

async def getHelloMessage():

    return { "Message": "Hello" };

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Run The Server That Make FastAPI Work With Async-Capabilities:

* **Run Command In CMD**: *uvicorn python-file:fast-api-app*
  + **Ex**: uvicorn main:app
  + **Note**: In Development, For Each Change: *uvicorn main:app --reload*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For Defining The Shape Of Models That We Can Use In Our App, We Need BaseModel From pydantic-Module:

from pydantic import BaseModel

# This Is For Our Request Content From User

class UserPostIn(BaseModel):

    body: str

# This Is For Our Output Response For User

class UserPost(UserPostIn):

    id: int

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Define The Shape Of Response Model, We Can Use: response\_model-attribute:

post\_table = {}

@app.post("/post", response\_model=UserPost)

async def create\_post(post: UserPostIn):

    data = post.model\_dump()

    last\_record\_id = len(post\_table)

    new\_post = {\*\*data, "id": last\_record\_id}

    post\_table[last\_record\_id] = new\_post

    return new\_post

@app.get("/post", response\_model=list[UserPost])

async def get\_all\_posts():

    # return post\_table.values()

    # OR We Can Use

    return list(post\_table.values())

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For Adding The Routes Folder For Each Related Data; We Can Use: **APIRouter**

from fastapi import APIRouter

from models.post import UserPost, UserPostIn

router = APIRouter()

@router.get("/")

async def getHelloMessage():

    return {"Message": "Hello"}

post\_table = {}

@router.post("/post", response\_model=UserPost)

async def create\_post(post: UserPostIn):

    data = post.model\_dump()

    last\_record\_id = len(post\_table)

    new\_post = {\*\*data, "id": last\_record\_id}

    post\_table[last\_record\_id] = new\_post

    return new\_post

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

async def get\_all\_posts():

    # return post\_table.values()

    # OR We Can Use

    return list(post\_table.values())

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then Inside The Main File That Use FastAPI-App Variable, And We Set The Prefix For The Route:

from fastapi import FastAPI

from routes.post import router as post\_router

app = FastAPI()

app.include\_router(post\_router, prefix="/post")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In This Way We Can Nest Models, To Use Each Other In Request OR Response Shape:

from pydantic import BaseModel

from post import UserPost

class UserCommentIn(BaseModel):

    body: str

    post\_id: int

class UserComment(UserCommentIn):

    id: int

class UserPostWithComments(BaseModel):

    post: UserPost

    comments: list[UserComment]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In This Way We Can Raise The Exceptions From FastAPI:

**Note**: The Details Can Be Dictionary That Contains Many Useful Information.

from fastapi import APIRouter, HTTPException

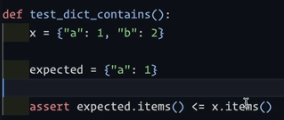
raise HTTPException(status\_code=404, details="Post Not Found")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For Adding ***The Status Code*** For Any Operation:

@router.post("/", response\_model=UserComment, status\_code=201)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Begin Testing The FastAPI-Client:

from typing import AsyncGenerator, Generator

import pytest

from fastapi.testclient import TestClient

from httpx import AsyncClient

from main import app

from routes.post import post\_table

from routes.comment import comment\_table

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Write The Test Client, We Use yield With The Function:

def client()-> Generator:

    yield TestClient(app=app)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Make Sure That Fixture Run With Every Test, We Can Use **autouse=True**:

@pytest.fixture(autouse=True)

async def db() -> AsyncGenerator:

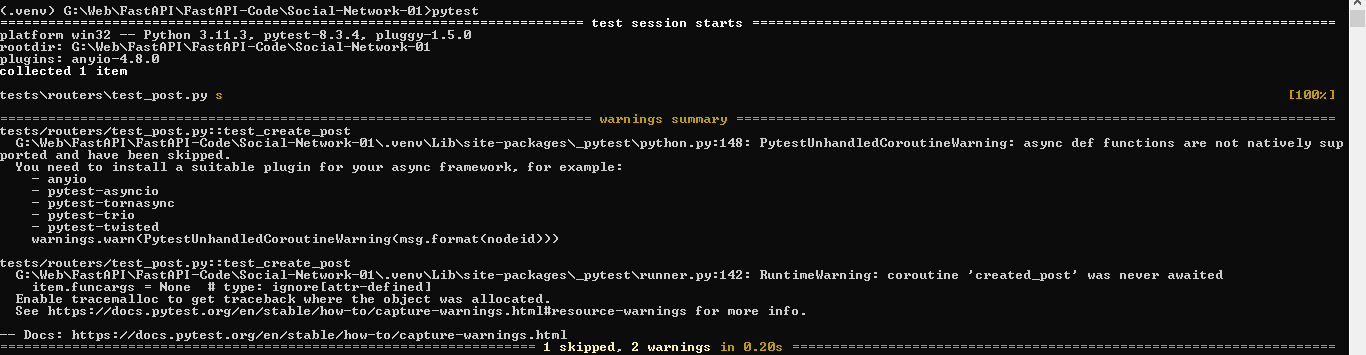
    comment\_table.clear()

    post\_table.clear()

    yield post\_table, comment\_table

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If We Run Pytest With Async/Await Then It Will Fail:



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Run The **Async/Await Code** With FastAPI, We Need To Mark Tests With: *@pytest.mark.anyio*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@pytest.mark.anyio

async def test\_create\_post(created\_post):

    assert created\_post[0] == 201

    assert {"body": "J-L-Test-01 Post"}.items() <= created\_post[1].items()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

But We Need To Define:

@pytest.fixture(scope="session")

def anyio\_backend():

    return "asyncio"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Note Here: If We Don't Set created\_post, Then The Test Will Fail

# Because db-fixture use autouse, so foreach test the db will be empty

# and no post will found

@pytest.mark.anyio

async def test\_create\_comment(created\_post: tuple, created\_comment: tuple):

    status\_code, data = created\_comment

    assert status\_code == 201

    assert { "body": "J-L-Test-01 Comment", "post\_id": 0}.items() <= data.items()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@pytest.fixture()

async def created\_comment(async\_client: AsyncClient):

    return await create\_comment("J-L-Test-01 Comment", 0, async\_client=async\_client)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

async def create\_comment(body: str, post\_id: int, async\_client: AsyncClient):

    response = await async\_client.post(url='/comment/', json={"body": body, "post\_id": post\_id})

    # For Debugging Only

    # print("The Status Code Is: ", response.status\_code)

    # print("The Response Body Is: ", response.json())

    return response.status\_code, response.json()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Make Pydantic Read The Configuration From .env Variables File:

* **Run The Command**: *pip install pydantic-settings*

Then We Define Our Config Settings Class That Represent The Main Class For OUR Configuration Classes That Are Used For Our Project

from pydantic\_settings import BaseSettings, SettingsConfigDict

class BaseConfig(BaseSettings):

    ENV\_STATE: Optional[str] = None

    model\_config = SettingsConfigDict(env\_file='.env', extra="ignore")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then To Define OUR Configuration Classes:

class GlobalConfig(BaseConfig):

    DATABASE\_URL: Optional[str] = None

    DB\_FORCE\_ROLL\_BACK: bool = False

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class DevConfig(GlobalConfig):

    model\_config = SettingsConfigDict(env\_prefix='DEV\_')

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class ProdConfig(GlobalConfig):

    model\_config = SettingsConfigDict(env\_prefix='PROD\_')

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class TestConfig(GlobalConfig):

    # If We Want To Override These Variables We Can Add Them

    # To .env-File

    DATABASE\_URL: Optional[str] = "sqlite:///test.db"

    DB\_FORCE\_ROLL\_BACK: bool = True

    model\_config = SettingsConfigDict(env\_prefix='TEST\_')

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then To Get The Values Of Our Class Depending On Env State:

**Note**: To Cache Functions Depending On Parameters, We Can Use lru cache from *functools-module*.

from functools import lru\_cache

@lru\_cache()

def get\_config(env\_state: str):

    configs = {"dev": DevConfig, "prod": ProdConfig, "test": TestConfig}

    if env\_state not in configs.keys():

        raise Exception("Invalid Value For env\_state Variable")

    return configs[env\_state]()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then To Get The Object Of Our Configuration:

config = get\_config(BaseConfig().ENV\_STATE)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Create The Database, With Its Tables, We Need MetaData-Object:

import databases

import sqlalchemy

from config import config

metadata = sqlalchemy.MetaData()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then We Create The Tables That We Need, And Combine Them With MetaData-Object.

First, It’s The Name Of Table.

Second, It’s The MetaData-Object.

Third, A List Of Columns That Table Contain, Without []

### Create The Posts Table ###

post\_table = sqlalchemy.Table(

    "posts",

    metadata,

    sqlalchemy.Column("id", sqlalchemy.Integer, primary\_key=True),

    sqlalchemy.Column("body", sqlalchemy.String),

)

### Create The Comments Table ###

comments\_table = sqlalchemy.Table(

    "comments",

    metadata,

    sqlalchemy.Column("id", sqlalchemy.Integer, primary\_key=True),

    sqlalchemy.Column("body", sqlalchemy.String),

    # Here We Don't Need To Tell The Type Of Column, Because It's ForeignKey

    # So It Will Give It The Same Type Of Id Of Posts Table.

    sqlalchemy.Column("post\_id", sqlalchemy.ForeignKey("posts.id"),nullable=False, ),

)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then We Will Create The Engine That Will Make The Database With MetaData-Object:

# connect\_args = { "check\_same\_thread": False } --> This Only Required For Sqlite

engine = sqlalchemy.create\_engine(

    url = config.DATABASE\_URL, connect\_args={ "check\_same\_thread": False }

)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Tell The Engine To Use The MetaData Object

# So, When We Run The Code, All The Tables That Are Combine

# With The metadata-object, Will Be Created

metadata.create\_all(engine)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

database = databases.Database(

    url=config.DATABASE\_URL, force\_rollback=config.DB\_FORCE\_ROLL\_BACK

)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In Python **ContextManager** Do The *Setup And TearDown*, Like We Use with-keyword.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Use The ContextManager With FastAPI LifeSpan:

**Note**: The Old Way Of Using Starup And Down With FastAPI Can Be Used Also, But This Is The Perfect Way Now.

from contextlib import asynccontextmanager

from database import database

@asynccontextmanager

async def lifespan(app: FastAPI):

    await database.connect()

    yield

    await database.disconnect()

app = FastAPI(lifespan=lifespan)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

from contextlib import asynccontextmanager

from fastapi import FastAPI

from routes.post import router as post\_router

from routes.comment import router as comment\_router

from database import database

@asynccontextmanager

async def lifespan(app: FastAPI):

    await database.connect()

    yield

    await database.disconnect()

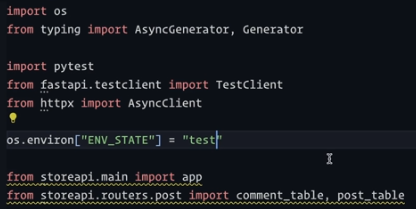
app = FastAPI(lifespan=lifespan)

app.include\_router(post\_router, prefix="/post")

app.include\_router(comment\_router, prefix="/comment")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The Best Way To Use Config In Test Environment, Before We Load Our App, We Override The **ENV\_STATE-Variable**:



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@pytest.fixture(autouse=True)

async def db() -> AsyncGenerator:

    # comment\_table.clear()

    # post\_table.clear()

    await database.connect()

    yield

    await database.disconnect()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*