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]
```
****

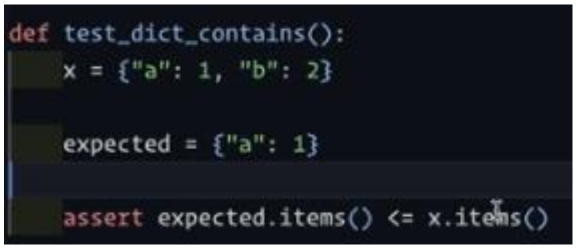
In This Way We Can Raise The Exceptions From FastAPI:

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)
```
****



```python
def test_dict_contains():
    x = {"a": 1, "b": 2}

    expected = {"a": 1}

    assert expected.items() <= x.items()
```

****

To Begin Testing The FastAPI-Client:

```python
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:

```python
def client()-> Generator:
    yield TestClient(app=app)
```

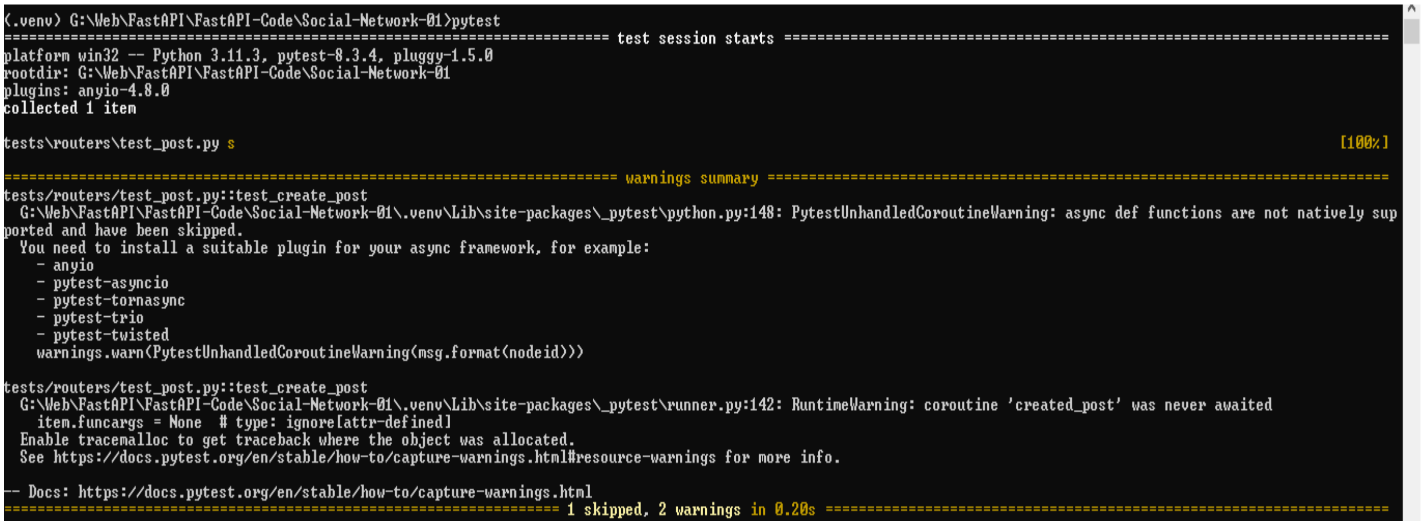
****

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

```python
@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:



```
(.venv) G:\Web\FastAPI\FastAPI-Code\Social-Network-01>pytest
============================================================================ test session starts ============================================================================
platform win32 -- Python 3.11.3, pytest-8.3.4, pluggy-1.5.0
rootdir: G:\Web\FastAPI\FastAPI-Code\Social-Network-01
plugins: anyio-4.8.0
collected 1 item

tests\routers\test_post.py s                                                                                                                                         [100%]

============================================================================= warnings summary ==============================================================================
tests/routers/test_post.py::test_create_post
  G:\Web\FastAPI\FastAPI-Code\Social-Network-01\.venv\Lib\site-packages\_pytest\python.py:148: PytestUnhandledCoroutineWarning: async def functions are not natively sup
ported and have been skipped.
  You need to install a suitable plugin for your async framework, for example:
    - anyio
    - pytest-asyncio
    - pytest-tornasync
    - pytest-trio
    - pytest-twisted
    warnings.warn(PytestUnhandledCoroutineWarning(msg.format(nodeid)))

tests/routers/test_post.py::test_create_post
  G:\Web\FastAPI\FastAPI-Code\Social-Network-01\.venv\Lib\site-packages\_pytest\runner.py:142: RuntimeWarning: coroutine 'created_post' was never awaited
    item.funcargs = None  # type: ignore[attr-defined]
  Enable tracemalloc to get traceback where the object was allocated.
  See https://docs.pytest.org/en/stable/how-to/capture-warnings.html#resource-warnings for more info.

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
====================================================================== 1 skipped, 2 warnings in 0.20s =======================================================================
```

****

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_')
```
****

```python
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*.

```python
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:

```python
config = get_config(BaseConfig().ENV_STATE)
```
****

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

```python
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.

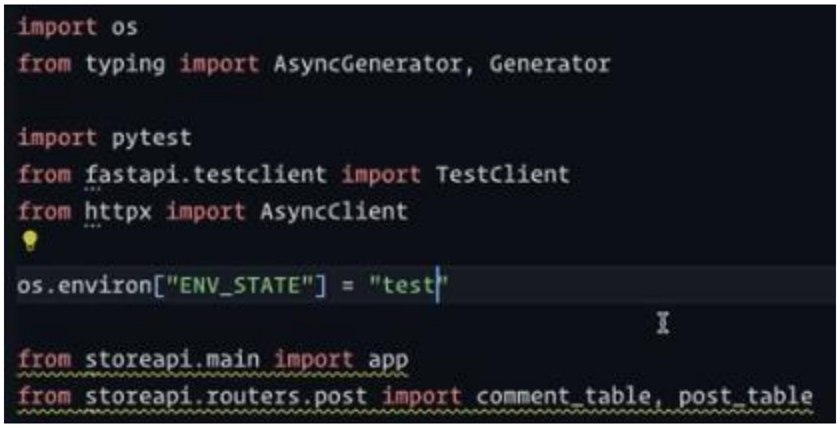
```
from contextlib import asynccontextmanager
from database import database

@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:



```
import os
from typing import AsyncGenerator, Generator

import pytest
from fastapi.testclient import TestClient
from httpx import AsyncClient

os.environ["ENV_STATE"] = "test"

from storeapi.main import app
from storeapi.routers.post import comment_table, post_table
```

****

```
@pytest.fixture(autouse=True)
async def db() -> AsyncGenerator:
    # comment_table.clear()
    # post_table.clear()
    await database.connect()
    yield
    await database.disconnect()
```

****