To Define The Endpoint That Make The User Create The Token For Make Authenticated Requested.

So, It Is Populates The Documentation With Authenticated Token, And Grab The Token From URL.

```
from fastapi.security import OAuth2PasswordBearer

oauth2_schema = OAuth2PasswordBearer(tokenUrl='/user/token')
**************************************************************************
```

To Use The Token Authentication For Protect Posts Endpoints, We Can Do:

```
from social_network.models.user import User

from social_network.security import get_current_user, oauth2_schema

@router.post("/", response_model=UserPost, status_code=201)
async def create_post(post: UserPostIn, request: Request):

    # In This Way We Protect The Endpoint From Un-Authenticated Requests
    current_user: User = await get_current_user(await
oauth2_schema(request=request))

    data = post.model_dump()

    query = posts_table.insert().values(data)

    logger.debug(f"The Query For Create Post Is: {query}")

    last_id = await database.execute(query)

    return  {**data, "id": last_id}
**************************************************************************
```

For Testing We Need To Define This Fixture, To Get Token, To Use It In Creating Post, And Creating Comment:

```
@pytest.fixture()
async def get_token(async_client: AsyncClient, registered_user: dict) -> str:
    response  = await async_client.post(url='/user/token', json=registered_user)

    return response.json()["access_token"]
**************************************************************************
```

To Use Dependency Injection For Getting User Data From Token:

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

oauth2_schema = OAuth2PasswordBearer(tokenUrl='user/token')

# Annotated[str, Depends(oauth2_schema)] --> That Means The Token Type Is String
# And Will Be Populated From oauth2_schema
async def get_current_user(token: Annotated[str, Depends(oauth2_schema)]):
    try:
        payload = jwt.decode(token=token, key=config.SECRET_KEY,
algorithms=[config.ALGORITHM])

        email = payload.get('sub', None)
        if email is None:
            raise credentials_exception

        user = await get_user_by_email(email=email)

        if user is None:
            raise credentials_exception
        return user

    except ExpiredSignatureError as e:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Token Has Been Expired",
            headers={
                "WWW-Authenticate": "Bearer",
            },
        ) from e
    except JWTError as e:
        raise credentials_exception from e
```

****

Note 1: In The Previous Way, We Don't Need To Call oath2_schema(request) For Each Function.

Note 2: The Depends Function Will Get The Value Of Token Instead Of Calling The Function.

****

```python
@router.post("/", response_model=UserPost, status_code=201)
async def create_post(post: UserPostIn, current_user: Annotated[User,
Depends(get_current_user)]):

    data = post.model_dump()

    query = posts_table.insert().values(data)

    logger.debug(f"The Query For Create Post Is: {query}")

    last_id = await database.execute(query)

    return  {**data, "id": last_id}
      ------------------------------------------------------
@router.post("/", response_model=UserComment, status_code=201)
async def create_comment(comment: UserCommentIn, current_user: Annotated[User,
Depends(get_current_user)]):

    data = comment.model_dump()
    post = await find_post(data['post_id'])
    if post is None:
        raise HTTPException(status_code=404, detail="Post Not Found")

    query = comments_table.insert().values(data)

    logger.debug(f"The Query For Create Comment Is: {query}")

    last_id = await database.execute(query)

    return {**data, "id": last_id}
```

****

Note: The New Way To Use Dependency Injection With Depends Is By Using Annotated

****

To Use Authentication With Swagger API, We Can Do:

```
from fastapi import APIRouter, HTTPException, status, Form, Depends
from fastapi.security import OAuth2PasswordRequestForm

@router.post('/token', status_code=200)
async def login_user(
    username: Annotated[str, Form()],
    password: Annotated[str, Form()],
    grant_type: Annotated[str, Form()]):
    access_token = await authenticate_user(username, password)

    return {"access_token": access_token, "token_type": "bearer"}
      ----------------------------------------------------
@router.post('/token', status_code=200)
async def login_user(form_data:  Annotated[OAuth2PasswordRequestForm,
Depends()]):
    access_token = await authenticate_user(form_data.username,
form_data.password)

    return {"access_token": access_token, "token_type": "bearer"}
      ----------------------------------------------------
```

Note: The Previous Configuration, Because OAuth2 Send Data AS: `application/x-www-form-urlencode`

****

To Define The Likes Table:

```
### Create The Likes Table ###
likes_table = sqlalchemy.Table(
    "likes",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    # 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, ),
    sqlalchemy.Column("user_id", sqlalchemy.ForeignKey("users.id"),
nullable=False),
)
**********
```

To Make Outer Join Between Posts Table And Likes Table:

```
select_post_and_likes = (
    sqlalchemy.select(
        posts_table,
        # .label('likes') --> Is Similar To Use: AS-Keyword
        sqlalchemy.func.count(likes_table.c.id).labale("likes"),

    )\
    .select_from(posts_table.outerjoin(likes_table))\
    .group_by(posts_table.c.id)
)
**************************************************************************
```

The Model Changes:

```
from pydantic import BaseModel

class PostLikeIn(BaseModel):
    post_id: int

class PostLike(PostLikeIn):
    id: int
    user_id: int
      ----------------------------------------------------------------

from pydantic import BaseModel, ConfigDict

# This Is For Our Request Content From User
class UserPostIn(BaseModel):
    body: str

# This Is For Our Output Response For User
class UserPost(UserPostIn):
    model_config = ConfigDict(from_attributes=True)
    id: int
    user_id: int

class UserPostWithLikes(UserPost):
    model_config = ConfigDict(from_attributes=True)
    likes: int
      ----------------------------------------------------------------
```

```python
class UserPostWithComments(BaseModel):

    post: UserPostWithLikes
    comments: list[UserComment]
```

In This Way We Can Select The Data From Posts, Comments, And Likes Table And Join Them.

****

```python
@router.get("/{post_id}/post-with-comments", response_model=UserPostWithComments)
async def get_post_with_comments(post_id: int):
    # post = await find_post(post_id)

    query = select_post_and_likes.where(posts_table.c.id == post_id)

    logger.debug(f"The Query For Getting Post, With Likes And Comments: {query}")

    post = await database.fetch_one(query)

    if post is None:
        raise HTTPException(status_code=404, detail=f"Post Not Found For Get Post
With Its Comments With Id: {post_id}")

    return {
        "post": post,
        "comments": await get_post_comments(post_id=post_id)
    }
```

****

In This Way, FastAPI Use Sorting As Query Parameter Because PostSorting Is Enum.

```python
class PostSorting(str, Enum):
    new = "new"
    old = "old"
    most_likes = "most_likes"

@router.get("/", response_model=list[UserPostWithLikes])
async def get_all_posts(sorting: PostSorting = PostSorting.new):
    # return post_table.values()
    # OR We Can Use
    query = select_post_and_likes.order_by(sqlalchemy.desc("likes"))
    logger.debug(f"The Query For Get All Posts Is: {query}")
    return await database.fetch_all(query)
```

****

If We Have Column Object Inside The Table, Then We Can Use Desc-Method Of It:

```
if sorting.new == PostSorting.new:
      query = select_post_and_likes.order_by(posts_table.c.id.desc())
elif sorting.old == PostSorting.old:
      query = select_post_and_likes.order_by(posts_table.c.id.asc())
```

****

If We Don't Have Column Object, But We Know The Column Name, Then We Can use *sqlalchemy.desc("column-name-here")*

```
query = select_post_and_likes.order_by(sqlalchemy.desc("likes"))
```

****

If We Pass Any Other Value For Sorting (Right Values Is: New, Old, Most_Likes), Then FastAPI Will Return Response With Status Code Is: 422

****

To Create Multiple Tokens For Using In (Login, Confirmations, Roles, ...etc) We Can Add type-Key To Data Of Token

Note 1: The *type: access* Means It is Used For Login

****

In This Way, We Can Getting The Details Of Exception That Raises From Test:

```
def test_get_subject_for_token_type_expired(mocker):
    mocker.patch("storeapi.security.access_token_expire_minutes", return_value=-1)
    email = "test@example.com"
    token = security.create_access_token(email)
    with pytest.raises(security.HTTPException) as exc_info:
        security.get_subject_for_token_type(token, "access")
    assert "Token has expired" == exc_info.value.detail
```

****

In This Way, We Can Build The URL For Specific Point:

Note 1: The *confirm_user_email* Is The Method Name For The Route.

```
return {
        "msg": "User Created Successfully",
        "id": result,
        "confirmation_url": request.url_for(
            "confirm_user_email",
            token=create_confirm_token(user.email)
        )
    }
```

****

```
@router.get("/confirm/{token}")
async def confirm_user_email(token: str):
    email = get_subject_for_token_type(token=token, token_type='confirmation')

    query = users_table.update().where(users_table.c.email ==
email).values(confirmed=True)

    logger.debug(f"The Query For User Confirmation Is: {query}")

    await database.execute(query)

    return { "detail": "User Confirmed" }
```

****

In This Way We Can Spy On The Value Of Request-Class From FastAPI-Module

Here We Spy Only, Not Change The Value.

We Return The Value Using **spy_return**

```python
@pytest.mark.anyio
async def test_confirm_user(async_client: AsyncClient, mocker):
    spy = mocker.spy(Request, "url_for")
    await register_user(async_client, "test@example.net", "1234")
    confirmation_url = str(spy.spy_return)
    response = await async_client.get(confirmation_url)

    assert response.status_code == 200
    assert "User confirmed" in response.json()["detail"]
```

****

```python
from unittest.mock import AsyncMock, Mock
```

```python
from httpx import AsyncClient, Request, Response
```

```python
@pytest.fixture(autouse=True)
def mock_httpx_client(mocker):
    mocked_client = mocker.patch("storeapi.tasks.httpx.AsyncClient")

    mocked_async_client = Mock()
    response = Response(status_code=200, content="", request=Request("POST", "//"))
    mocked_async_client.post = AsyncMock(return_value=response)
    mocked_client.return_value.__aenter__.return_value = mocked_async_client
```

****

Note 1: Here We Must Set Empty Line Between *From* And The *Body*

```
async def send_simple_email(to: str, subject: str, body: str, from_: str):
    sender = f"Private Person <{from_}>"
    receiver = f"A Test User <{to}>"

    message = f"""\
Subject: {subject}
To: {receiver}
From: {sender}

{body}
"""


    with smtp.SMTP(config.EMAIL_HOST, config.EMAIL_PORT) as server:

        server.ehlo()

        server.starttls()

        server.login(config.USERNAME, config.PASSWORD)

        result = server.sendmail(sender, receiver, message)
**************************************************************************
```