⇒ Now, we will see the logic of verification of JWT whether it is not-tampered, or is not expired

⇒ first thing to do is to setup a Pydantic model (schema) as to specify the format for sending access_token & token_type in the request.
After login, for every action that needs user to be logged in, for every such request, access_token & token_type must be sent.

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

⇒ Also, we create a Pydantic schema model, for the data embedded in token :-

```
class TokenData(BaseModel):
    id: Optional[str] = None
```

7:15:40

⇒ In the oauth2.py file, we create a function to verify the access token :-

```
def verify_access_token(token: str, credentials_exception):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms =[ALGORITHM])
        user_id: str = payload.get("user_id")

        if id == None:
            raise credentials_exception
        token_data = schemas.TokenData( id = user_id )
    except JWTError:
        raise credentials_exception
    return token_dat
```

user_id

must be passed as a list, for decode() method

- In the JWT verification function, we create 2 arguments — to take the token (as string), and an object for credentials exception.

- We keep main coding of this function, inside a try block, as any exception occurring in it, can be handled, by the type JWTError class.

- In case of any exception, we will use the object obtained as 2nd parameter as follows:-
  raise credentials_exception

- We at beginning, recover the data, encoded in the payload component of JWT. It is done with the jwt.encode() method.
  It takes 3 argument :- i) Token (as string), ii) SECRET_KEY, iii) algorithm used in signing.

Returns the payload data as Python dict type.

From it, we extract the data of our choice. Here, we are extracting the value corresponding to the key user_id by this :- user_id = payload.get("user_id")

If user_id is none, then we raise exception. Else, we create an object of Pydantic schema class TokenData type, that we made.

- If token is valid, then token_data is returned.

Now, we need to create another function, that we will name as get_current_user

We will pass this function as a dependency, to any of the path operation, that requires validation of JWT.

It will take the token from the request automatically, will verify the token by calling the above verify_access_token function, extract the id from it.

Then, if we want, we can automatically fetch the user from the database, and add it as param to our path operations.

For the function that we are going to make, we need a dependency, that uses following oauth2_scheme

```
from fastapi.security import OAuth2PasswordBearer
oauth2_scheme = OAuth2PasswordBearer(...
                    ...tokenUrl = "login")
```

this is the endpoint of API, where users login, and then the JWT token is created & used further for authentication

Now, our get_current_user() function will be as follows:-

```
def get_current_user( token: str = Depends(oauth2_scheme) ):
    credentials_exception = HTTPException(
        status_code = status.HTTP_401_UNAUTHORIZED,
        detail = "Could not validate credentials.",
        headers = {"WWW-Authenticate": "Bearer"}
        )

    return verify_access_token( token, credentials_exception )
```

⇒ Here, we accept the JWT token as parameter, and type define it using dependency: Depends(oauth2_scheme)

⇒ We then define the exception credentials_exception that will be raised in case of invalid credentials in JWT
It is an HTTPException with :-
- status_code as 401 - Unauthorised.
- headers - a dictionary as above.
- A suitable error message in detail.

⇒ In return statement, we call the token verification function, by passing the token & credentials_exception

---

Mandating login for various path operations

⇒ We have our functions ready. Now, for the path operations, for which we want users must be logged in to do those, activities, we want to add an extra parameter to those path operation functions as follow:-

```
from <path> import oauth2
...get_current_user: int = Depends(oauth2.get_...
                        ...current_user)...
```

the function we created above

For eg:-

Path operation for creating posts:-

```
@router.post("/", status_code=201, ...)
def create_posts(post: schemas.PostCreate, d: Session,
    ... get_current_user: int = Depends(oauth2.get_current_user)):
    - — —
    — — —
```

3rd parameter of this path operation

⇒ Thus, for this path operation endpoint to be used, user must send access token in request, and that can only happen when user is logged in.

⇒ The dependancy calls the oauth2.get_current_user function, and in the function, automatically passes the JWT as parameter, and then whole set of functions get executed as per flow.