6:33:00

AUTHENTICATION :-

There are 2 main ways to handle authentication when working with API's & applications :-

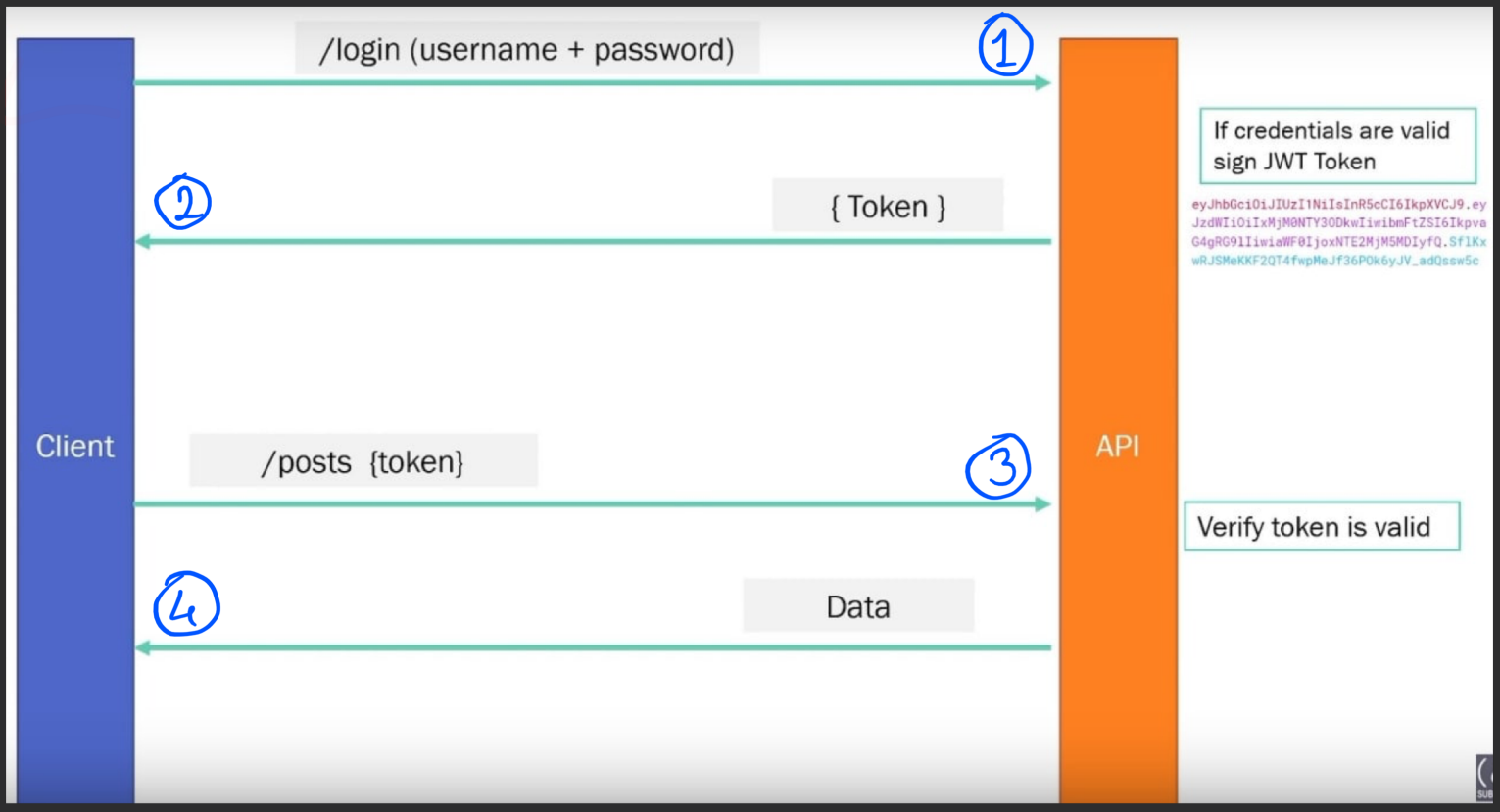
i) Session-Based Authentication :- Concept of session is that we store some data on backend server to know if user is logged in. This data stored (in db or in memory) track when user logs in & logs out.

ii) JWT-Token Based Authentication :- It is a stateless authentication — means that there is nothing stored in memory, db or API, that tracks logging activity of users.
Instead, "JWT token (that tracks if user is logged in) is stored on client-side / frontend.

Flow of Authentication Process :-

There are certain series of processes that takes place during authentication process.

1. User sends login request to login endpoint of API (generally /login), by passing required credentials like email/username & password



At backend, it is checked that whether such user exists and the credentials are True. If they match, user is successfully validated, and a JWT token specific to that user is generated.
[JWT token is a string that contains embedded information]

(2) As a response to login, the JWT is sent back to the client, and it gets stored on client's end. Using this JWT, client can start accessing resources that requires authentication.

(3) When user tries to access resources that require user to be logged in, with the request (to resource endpoint), user sends that stored unique JWT token. → /posts {token}.
Token is passed in the header of the request.

(4) The API then validates the token, and if the token is valid, the API sends the requested data to the client.
There are certain steps that is required to be done for token validation by API.

(*) Here, API does not at all, directly tracks user's logging activity.

JWT: deep dive :-

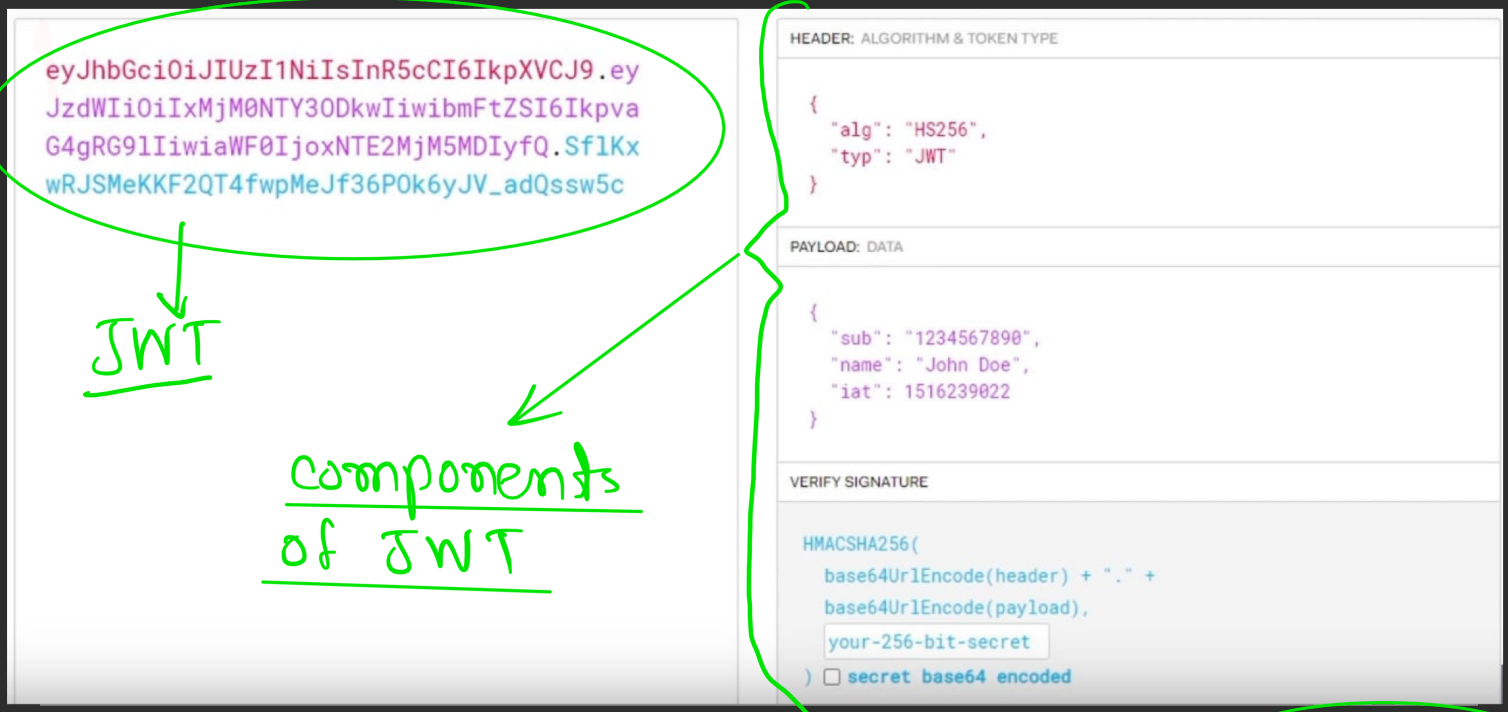
- JWT stands for JSON Web Token.
It is a JSON object which used to securely transfer information over the web.

- JWT is not encrypted.

- JWT is made of 3 individual parts/components
They are :- i) Header, ii) Payload, iii) Signature

i) Header :- Header contains metadata about the token.

- We actually sign the token, and algorithm used for it is mentioned in Header. e.g :- HS256.
- Also, it contains type of token. Here :- "JWT"



6:38:40

ii) Payload :- The data in payload can be anything that we want, but we should be careful that as JWTs are not encrypted, anyone can decipher the data in it. Thus, in payload, we must not send any confidential data.

Normally, data embedded in JWT payload include user id, user's role (whether admin or normal user), etc...

NOTE:-

We must not embed considerably large amount of data in payload section of JWT. This is because, whenever any action will be done needing authentication, user needs to send the JWT alongwith req. Large amt of data will cause JWT size getting bigger, and thus, will need more bandwidth.

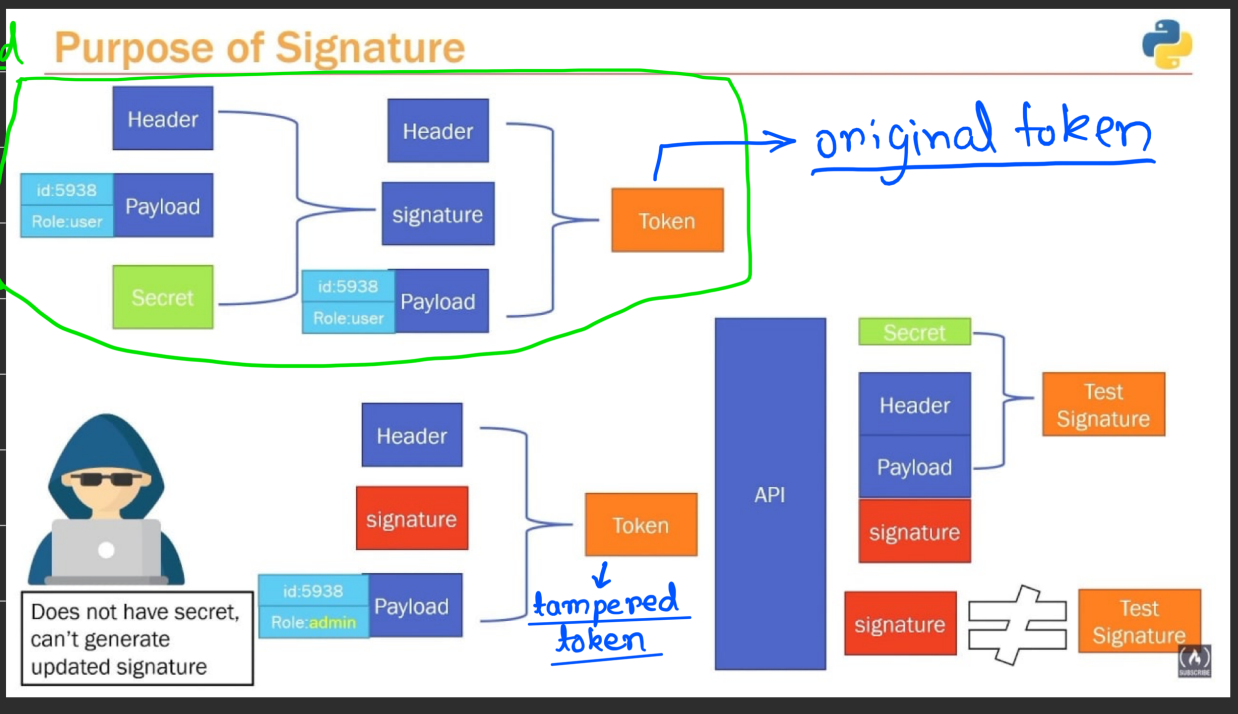
iii) **Signature:-** This is used to determine if the token is valid.

- The signature is a combination of 3 things:-
  - The header, already in the token
  - The payload, already in the token.
  - A special password, called **secret**, This is only kept on API, and clients could not know

These 3 things are taken and passed on to the signing algorithm (which is **HS256**), and in return, we will get the signature.

Thus, no one can tamper the token, as changing the token will change the signature
Thus, the signature maintains Data Integrity.

Purpose of Signature:-

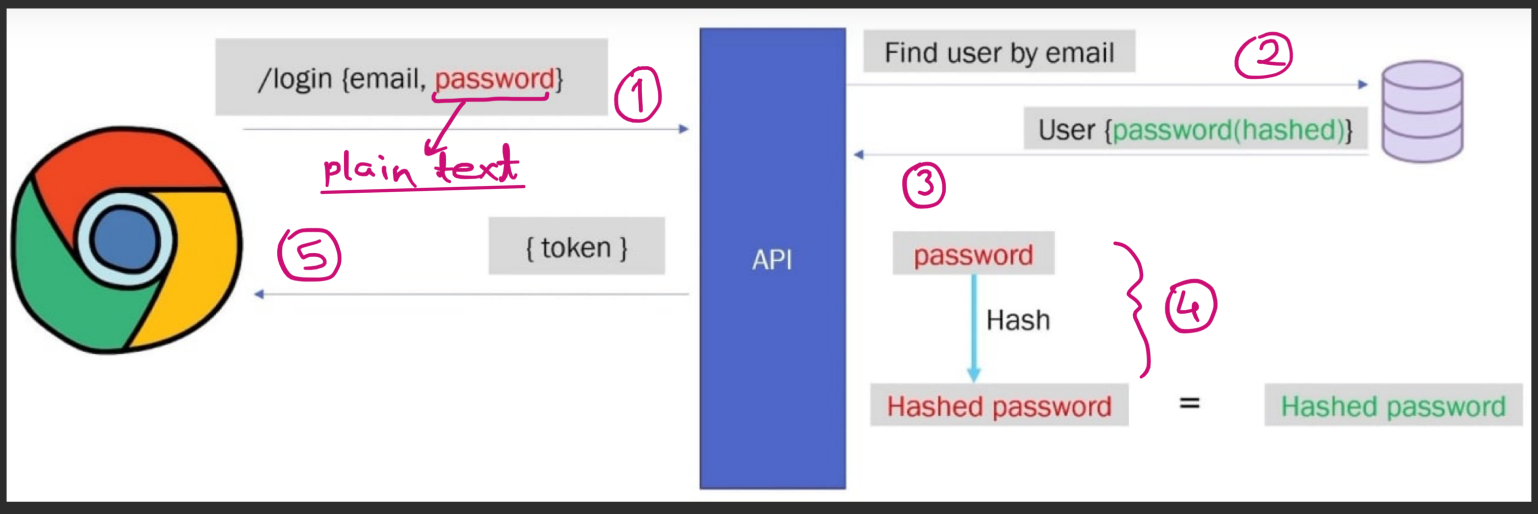


⇒ Now, if a hacker changes the payload data to some other user's data, then, the payload gets changed in his malicious JWT token, but still, that JWT token's signature remains same as he/she just can't change that.

so, at the API's backend, using the header & changed payload, (and secret), token is generated, and this will not match with the tampered token's original signature. Thus, it will verified as an invalid JWT token.

⇒ Hacker cannot create authentic signature, as he can only pass the header & payload to HS256 algorithm, but cannot pass the secret, as he does not have that.

⇒ Logging In User :-



i) In login attempt, user sends email & password as plain text.

ii) We find if any user with that email exist, and if we find that, we fetch the hashed password of the user from the database.

iii) We hash the plain text password (sent by user) and compare the obtained hash with the fetched hash from the database.

iv) If both hashes match, only then the user is validated, and a JWT token is then generated & sent to the user, in the response.

6:49:35

» Login Path Operation :-

» We create a seperate router (file), named auth.py, and authentication related path operations like login, will be coded here.

pydantic model for login request data

```python
@router.post("/login")
async def login(user_cred: schemas.UserLogin ,db: Session = Depends(database.get_db)):
    user = db.query(models.User).filter( models.User.email == user_cred.email ).first()
    if user == None:
        raise HTTPException( status_code = status.HTTP_404_NOT_FOUND, detail = "Invalid credentials!" )
    elif not utils.verify( user_cred.password, user.password ):
        # This is the case where email exists in database, but provided PASSWORD IS INCORRECT!
        raise HTTPException( status_code = status.HTTP_404_NOT_FOUND, detail = "Invalid Credentials!" )
    else:
        # Provided email and password are CORRECT
        # Generate & Return token
        # Return token
        return {"token": "example_token"}
```

Implementation of generation & sending of JWT here

testing purpose

This is our custom made function, taken takes plain text password as 1st param, & compares it with hashed password (2nd param) in the database.

⇒ Above is the logic for the login path operation. We just need to implent in it generation & sending of JWT token upon correct credentials being entered.

⇒ The utils.verify() method used above, is a follow:-

```
from passlib.context import CryptContext
pwd_context = CryptContext(schemes="bcrypt",
                           ... deprecated="auto")

def verify(plain_password, hashed_password):
    return pwd_context.verify(plain_password,
                              ... hashed_password)
```

inbuilt function is used, present in CryptContext object.

7:00:40

Handling generation of JWT:

Firstly :-

⇒ We need to install a library that handles signing and verifying JWT tokens. The library we will be using is python-jose

```
pip install python-jose[cryptography]
```

- We create a file named oauth2.py, in our main project folder (where main.py file exists)
Here, we will code JWT related functionalities

- For every JWT creation, we need to provide 3 things:
i) Secret Key, ii) Algorithm; (e.g.:- HS256)
iii) Expiration time of the token :- If not provided, JWT will remain valid forever.
- Function to generate token is as follows :-

```python
from jose import jwt, JWTError
from datetime import datetime, timedelta

# Details for the JWT generation
SECRET_KEY = "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

def create_access_token(data: dict):
    # Making a copy of the data that is passed as argument, and is to be encoded in the payload
    to_encode = data.copy()

    expire = datetime.utcnow() + timedelta( minutes = ACCESS_TOKEN_EXPIRE_MINUTES )
    to_encode.update( {"exp": expire} )

    # Generating the JWT token with all the information
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm = ALGORITHM )

    return encoded_jwt
```

- In SECRET_KEY, we need to assign a random String. It may be of any length, but for security purposes, it should be long enough.

- Mentioning the algorithm to be used for signing in the ALGORITHM variable.

- We need to also specify the time interval (in minutes) after which each JWT will get expired in the variable named ACCESS_TOKEN_EXPIRE_MINUTES

- Then, we create a function that can be used to generate token, by passing payload data as dictionary in the parameter.

We make a copy of the payload data, using copy() method, and store in a variable named to_encode. This copy is made, so that original data remains intact, when we make some changes to data in order to embed it into token.

- We create a time object, which contains the absolute time, at which the token will expire.
Obtained by adding ACCESS_TOKEN_EXPIRE_MINUTES interval to current time, and store in variable named expire.
We then add this expire object in the to_encode data dictionary, with suitable key (here, "exp")

- Finally, we create the JWT using jwt.encode() method. It takes 3 arguments & returns the token:-
1) The payload data, which is to_encode here.
2) The SECRET_KEY
3) The ALGORITHM as keyword argument.

- Then, we return this generated token.

Now, in our login() path operation, we can implement the access token code, as follow:-

```
async def login(...):
    ...
    access_token = oauth2.create_access_token(
        ... data = {"user_id": user.id})

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

Here, we can put any data on to the payload of the JWT.

token_type should be "bearer"

OAuth2PasswordRequestForm:-

Till now, the format of data in the login request is being maintained using Pydantic model schemas.UserLogin But now, in place of the Pydantic model, we will use OAuth2PasswordRequestForm for the format/type of the 1st argument containing user's credentials:

```
from fastapi.security import OAuth2PasswordRequestForm
...
def login(user_cred: OAuth2PasswordRequestForm = Depends()):
    user = db.query(models.User).filter(models....
        ...user.email = user_cred.username
```

- OAuth2PasswordRequestForm object contains two attributes :- username & password

They contain the username/email & password passed by user, respectively.

The, the user_cred object is like follows :-

```
{
  "username": ....,
  "password":
}
```

Thus, it is just like using Pydantic model with above fields.

username field can be email, username, or some id, or anything that is required for login.

- At parameter, OAuth2PasswordRequestForm must be assigned Depends() to build a dependency.

- Now, in the request, the credentials will not be accepted as JSON object in raw body.

Now, the credendentials username & password must be passed as form-data in Postman.

The keys of form must be username & password