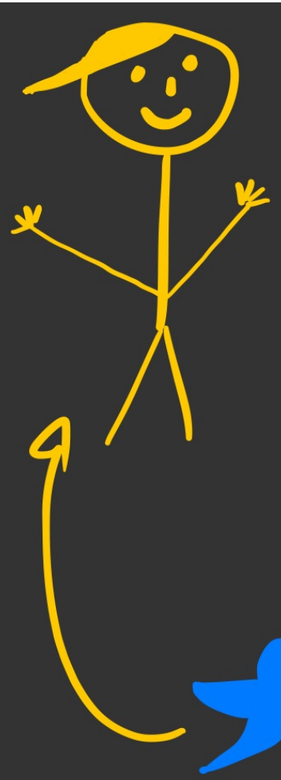
WEB AUTHENTICATION & AUTHORIZATION METHODS

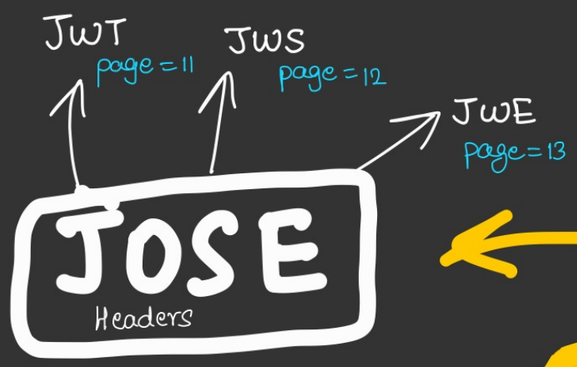


Hey, I am Rohit !!!

I know a few methods.

I can help you understand 'em

sec_r0



Page = 3

Basic Auth

Page = 5

Session based

I know Only these

* They are Sufficient to Understand for Now

Page = 7

Token based

Page = 17

OAuth

Authorization code

Page = 19

Implicit Code

Page = 21

Password grant

Page = 24

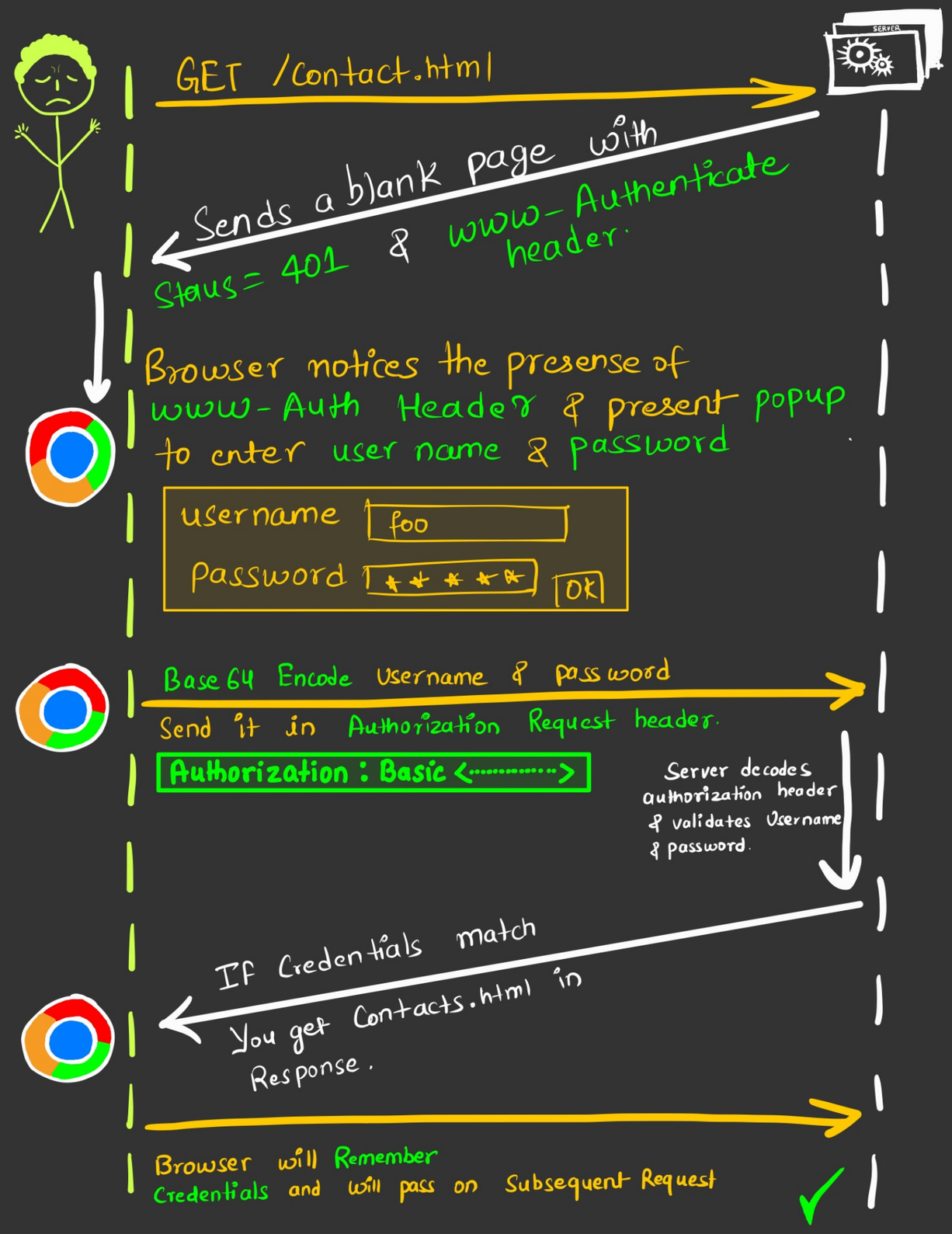
Client Credentials grant

Page = 23

Openid

page = 27

HTTP BASIC AUTHENTICATION



sec_r0

HTTP BASIC AUTH

☆ Credentials in header are base64 encoded not encrypted.

☆ Basic Auth with HTTP is not safe as data is sent in plaintext. I would suggest HTTPS if Basic Auth is the only option.

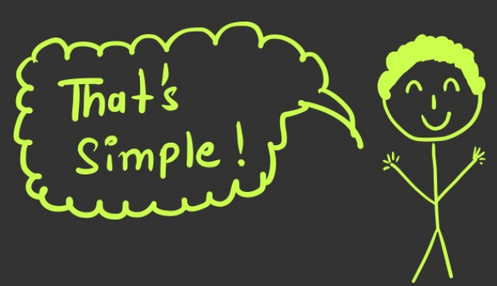
* Basic Auth Sends the Credentials as encoded text in Authorization Header. But it is a form of authentication not authorization

* Client (Browser) will ask for credentials if and only if Server sends back WWW-Authenticate Header in response. with Unauthorized Status 401

WWW-Authenticate : Basic realm = "foo" → Realm name.

Realm defines the scope (set of pages) where your credentials will be made available by client to server.

A Single site may have various realms



Session Based Authentication

USER login is linked with piece of **data** in server memory

POST { "username": "Jon", "Password": "FooBar" }

User Submit Credentials over POST

Server Validates User Credentials

Server identifies user and Sends browser a Cookie that will identify on future Request.

At this point Client is logged in the Server.

Server Stores Session in Memory or DB

Browser Saves Cookie and Send it to all Subsequent Requests (This is called Session Cookie)

GET /contacts.html

Sends Cookies as well

Server Validates the session information from Cookie to Session info from memory or DB.

Contacts.html

Status=200 OK

POST /logout.html

Invalidate Session token

Status=200 OK

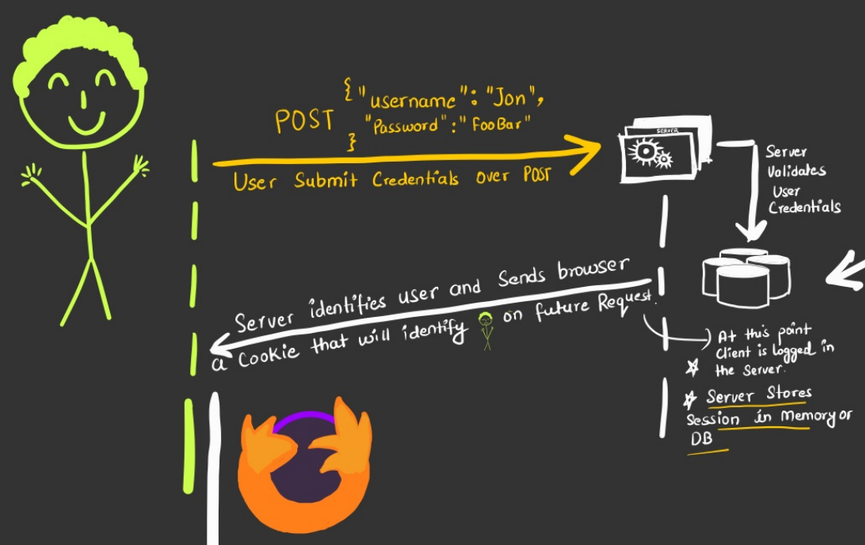
GET /Contacts.html

Status=403 User NOT Authenticated

The existing  has been invalidated, user unauthenticated.

sec_r0

Session Based Authentication



* server stores the session.

Browser Saves Cookie and send it to all Subsequent Requests. (This is called Session Cookie)

* User Login Session is stored in server memory.

* On each subsequent request server will compare session info present in cookies with session info in incoming requests.

* Till the user session is alive, server will have to preserve the session id.

* This approach does not scale well. As overhead on server increases.

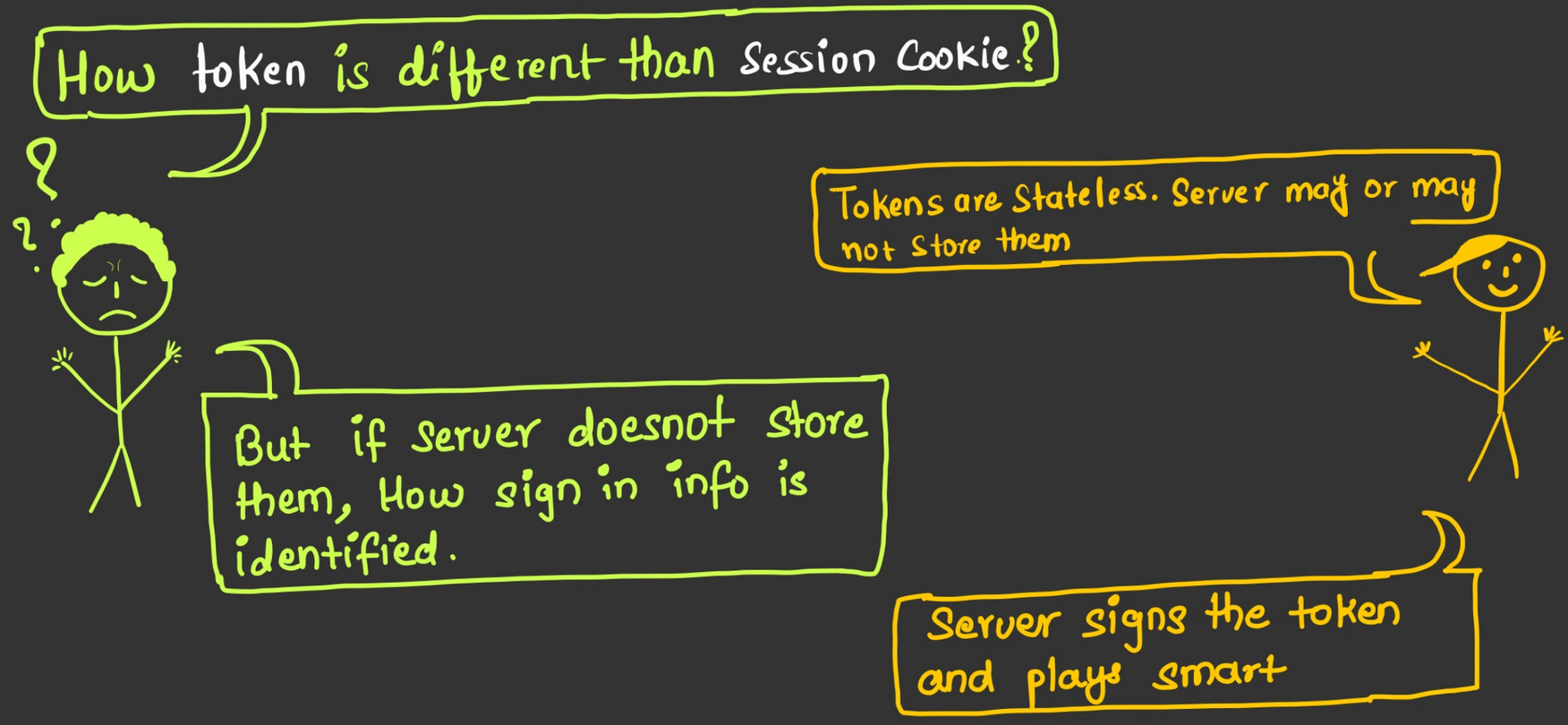
sec_r0

Token Based Sign In

- Token is small piece of data sent by server after successful login
- Future request to server will carry a token

Token based Sign in are popular with

- RESTful APIs
- Single Page Apps
- Interaction b/w Micro Services



Token Based Signin via

POST
```
{ "username": "Jon",
  "Password" : "FooBar"
}
```
User Submit Credentials over POST

SERVER

Server Validates User Credentials

Server identifies user and Sends browser a token that will allow access for user

At this point Client is logged in the Server.

Server Creates token and Signs it then sents it to client.

This is optional !

Client Saves token in local DB

Local browser storage

Browser Saves token and Send it with all Subsequent Requests.

GET /contacts.html

Sends Token as well

Server decodes the token & Validate Sign (if present)

Contacts.html

Status = 200 OK

POST /logout.html

Ask Client to clear token

Status = 200 OK

The flow can be Stopped here as few clients may not have capability to clear token

Clear data

GET /Contacts.html

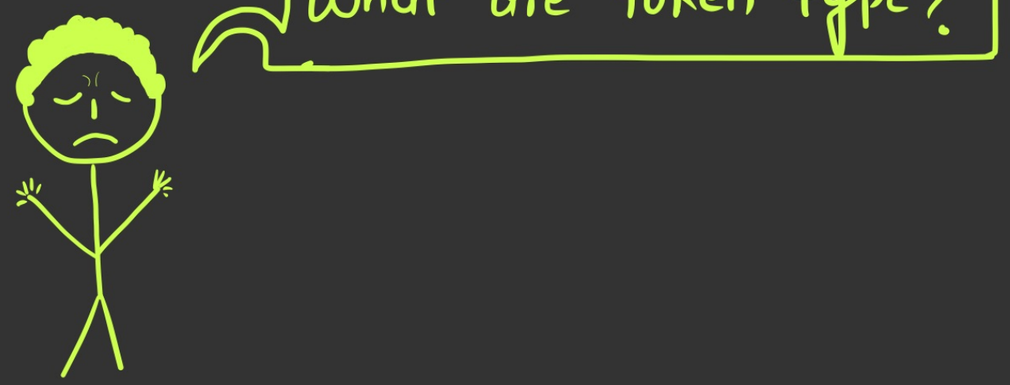
No token present.

Status = 403 User NOT Authenticated

sec_r0

Token Based Authentication

* Key Concept is tokens are Stateless
* Server should sign the token, so it may not need to store it on backend.
* Signature prevents tokens from tampering



*Also, Token based approach works cross domain too as tokens are sent via headers unlike session cookies which are bound to domains

Can You tell me about any of those tokens?

Needless to Say

JWT is really Important, I was just about to discuss them and SAML in another Zinc

But keep a smiling face & get ready to start.....

Why Not !

sec_r0

JWT

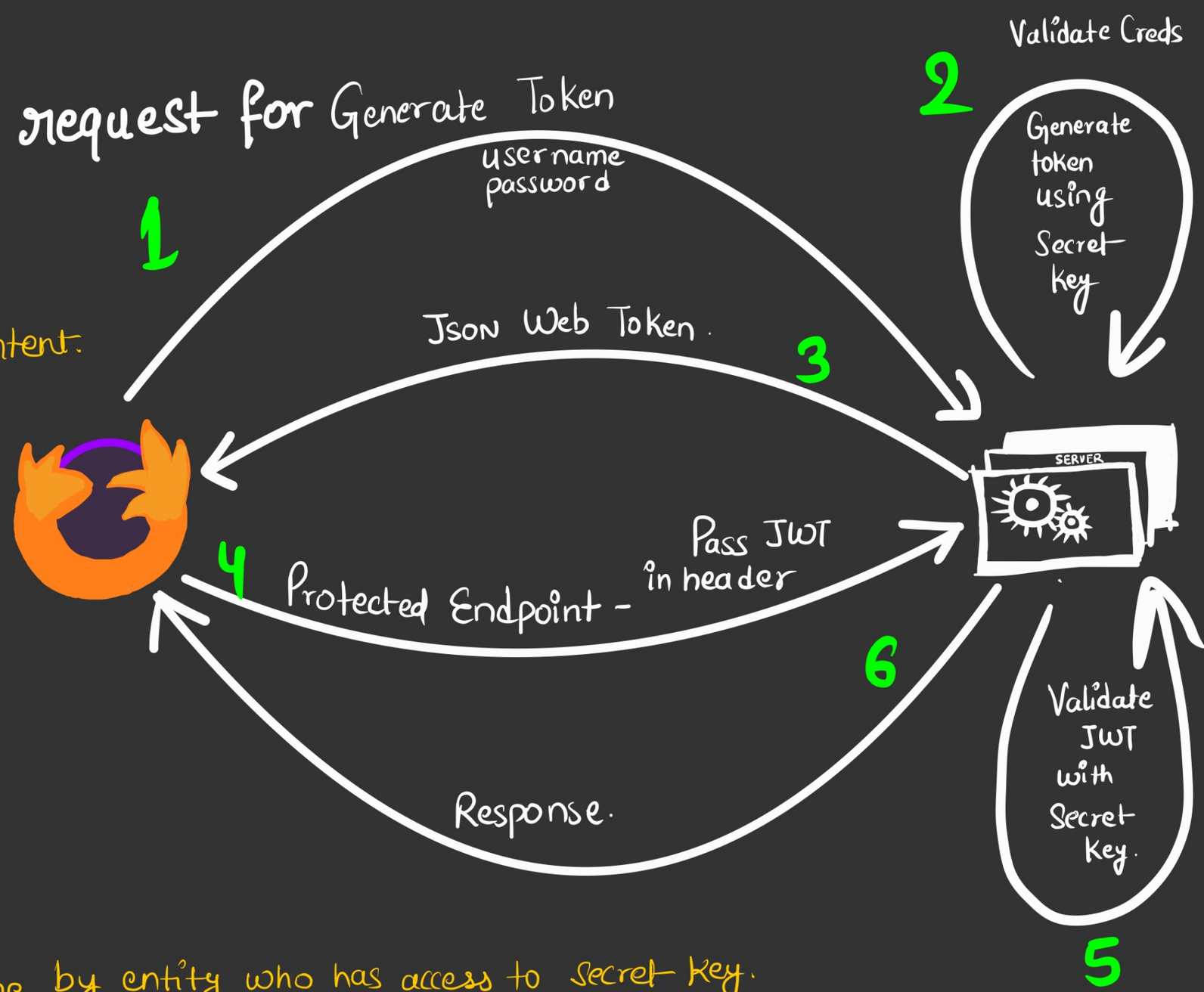
JSON WEB TOKENS

Can be used for token based authentication & may be also used for authorization
(Id token via OIDC)

* Self Contained :

Contains the data

*Anyone Can view the Content.



*verification can be only done by entity who has access to Secret key.

*Secret key here is a catch. If Asymmetric key Algorithm is used, Private key is used to sign JWT and anyone who have access to public key can verify.

sec_r0

JWT

JSON WEB TOKENS

3 parts Separated by dots

JWT

HHHHHHH . PPPPPPPP . SSSSSSSS

Optional

Header

Base64Encode(Token metadata)

```
{
  "typ" : "JWT"      → Token Type
  "alg" : "HS256"    → you can even use MAC or XML digital signatures.
}
```

Hashing Algorithm used for Signature

Payload

Base64Encode(data)

```
{                          DATA
  "userid" : "Jdoe",
  "email" : "john@doe.com",
  "exp" : "1234567890",
  "iat" : "78129401"
}
```

AKA CLAIMS

3 types

Public

which we Can define for own data. Means not officially Registered. often used by others.

Private

Name without meaning to anyone except token producer. They are also not registered and has not been previously defined.

Restricted

Name Reserved for App Usage.

Eg.

- **IAT** — issued at
- **ISS** — Issuer
- **Sub** — Token Subject
- **Exp** — expiry time
- **JTI** — JWT token Identifier

Signature

HMACSHA256(Header + '.' + Payload , Secret)

Secret → Held at Server

sec_r0

JWS

JSON WEB SIGNATURE

JWT

* In token based authentication, authentication server creates JWT token and gives it to client.

* There are two instances of JWT

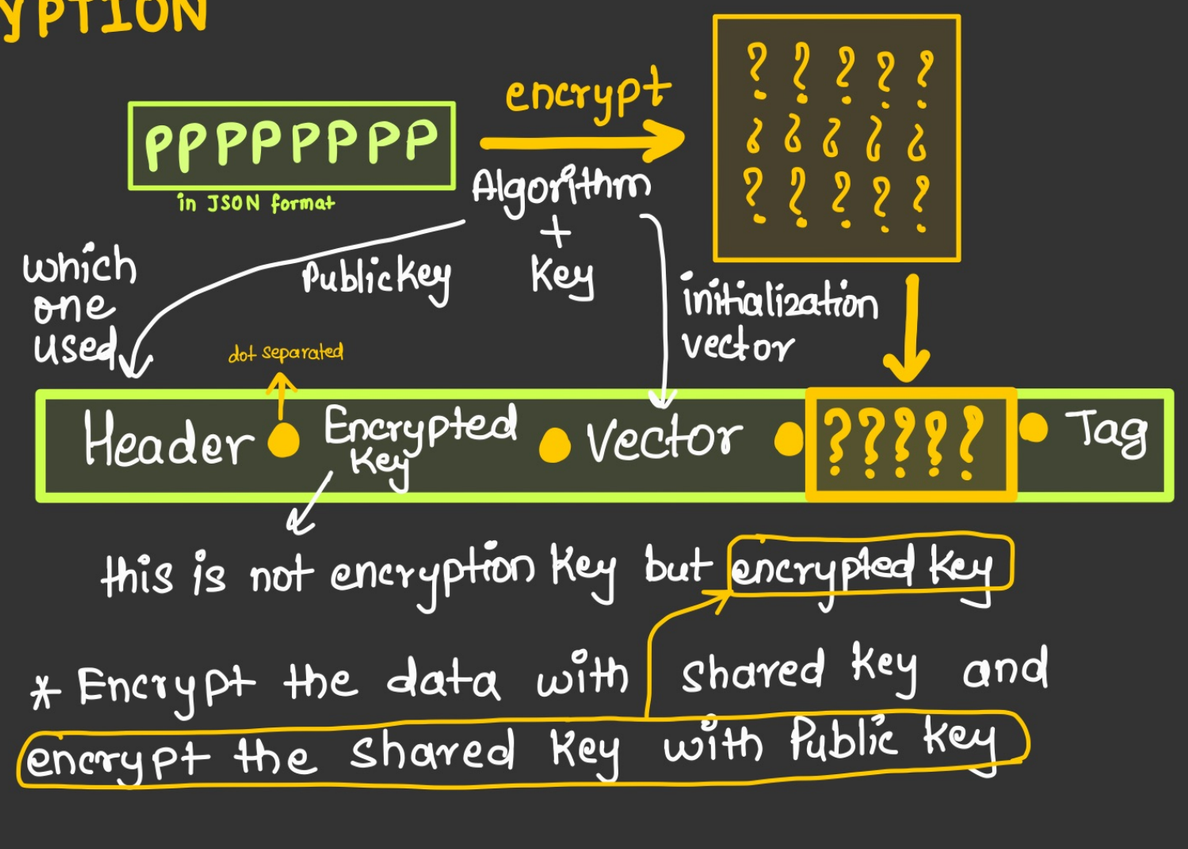
JWE Encryption JWS ignature [if optional signature is present they become JWS]

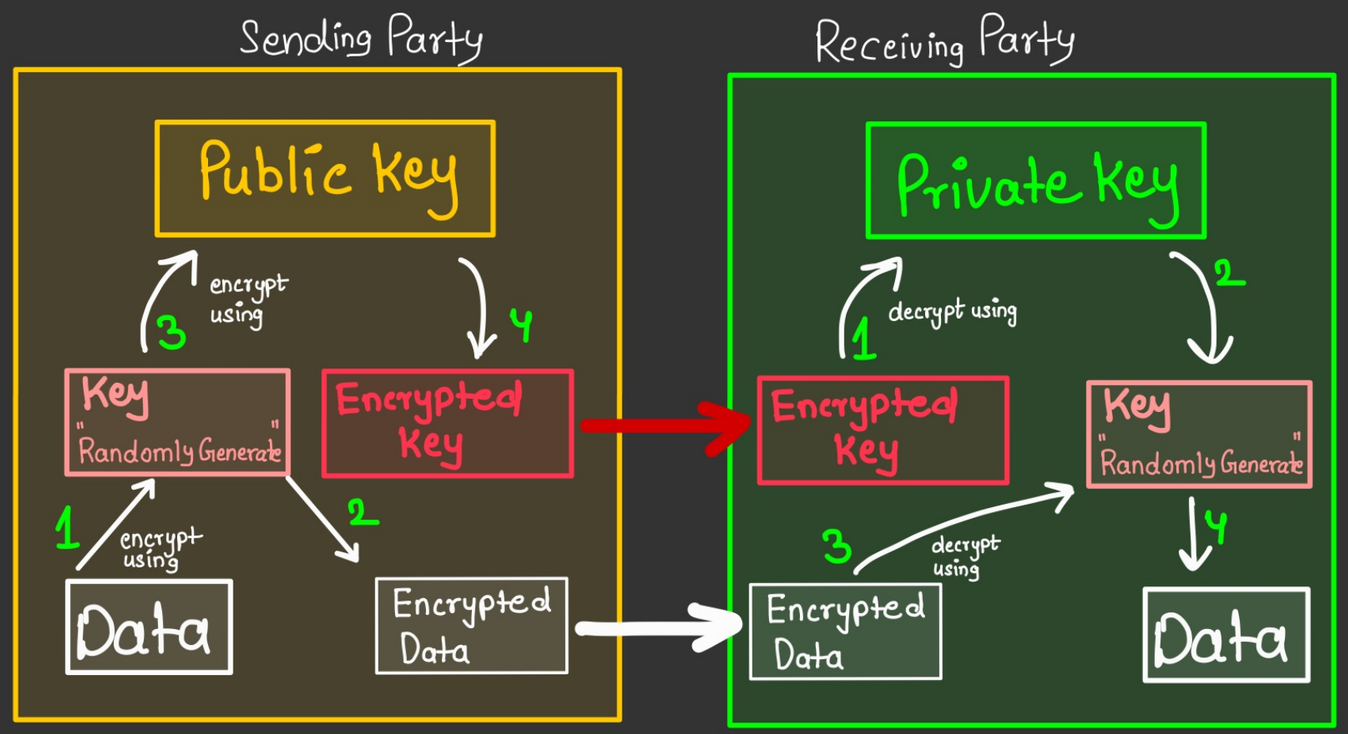
* The JWT we commanly use in OAuth is actually a JWS implementation but we mostly call it JWT.

* Rest all of the structure is same.

* So when we say JWT its mostly JWS.

JWE JSON WEB ENCRYPTION

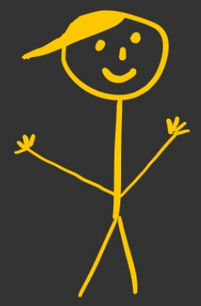




sec_r0

This was too much of knowledge!!

Agreed, take a break, refresh & then we will move to OAuth



I hope that will be easy.

Don't Worry I will make it Easy!

What is the difference between Authentication and Authorization?

* Authentication is validating user name and password of user.

* Authorization is process of giving user access to resource after authentication.

* Authorization protocols like OAuth which we are going to discuss next performs username and password check but since it doesnot share user identity with some one else, it is an authorization protocol, Not authentication.

* we can convert OAuth to Authentication by adding support for identity sharing and thats what OIDC does.

OIDC → open identity connect

sec_r0

OPEN AUTHORIZATION

After teaching Authorization ways, you are asking me to open !!!

What the Hell !!

Don't get Confused by name!

It Says "Open protocol for Authorization"

sec_r0

It allows the users to share their private resources without sharing username & password.

Rather the protocol generates access token which allows resource access for client

There are 4 Entities Involved

Client App

App Frontend runs here

This can be a webapp or stand alone application.

Authorization Server

Resource Token — generator, validator and Access gatekeeper

* may create access token in form of JWT and Sign it with its private key.

Resource Server

Actual protected Resource

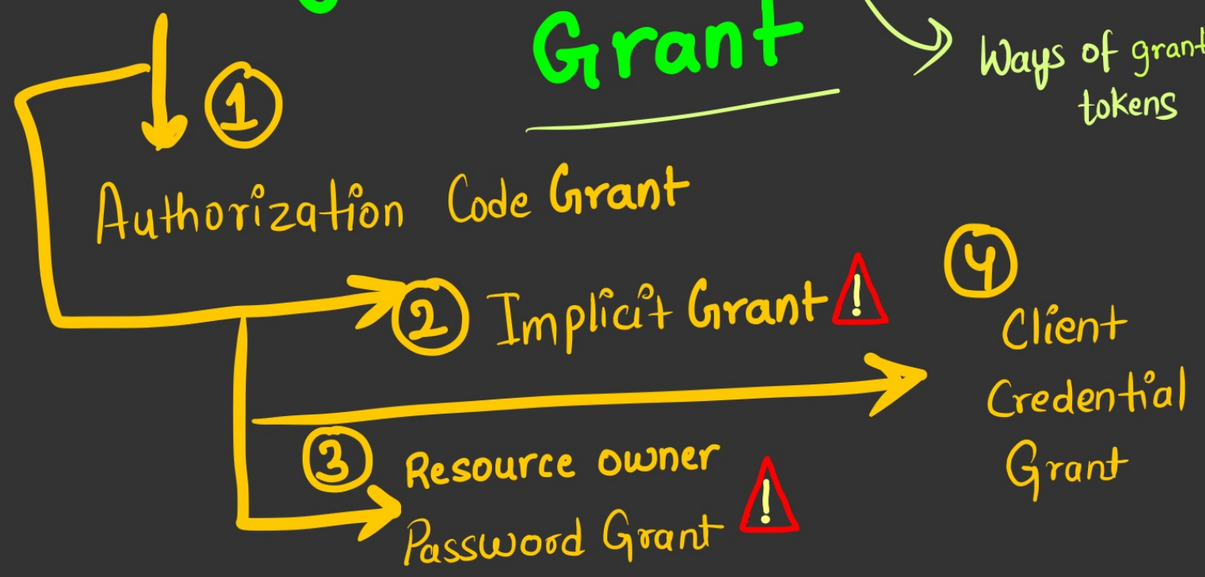
* Once the resource server receives access token, It validates access token, if that is Okay or not.
* In Case of JWT, signature can be used to validate signature.
* Authorization server may sign token with private key and resource server verifies with Authz server public key.

User

owner of Credentials

4 types of Authorization Grant

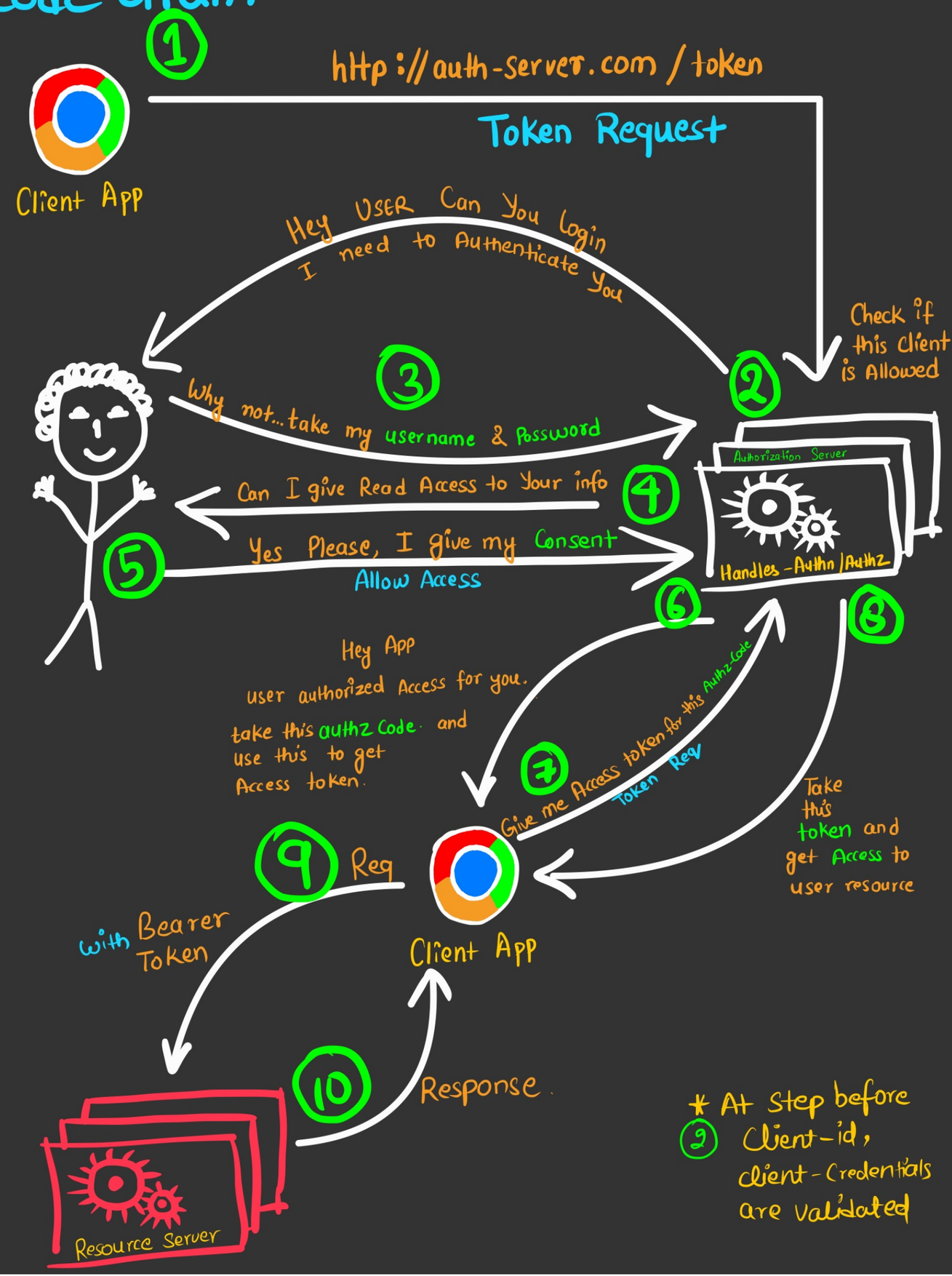
Ways of granting tokens



not used as they have security Concerns

sec_r0

Authorization Code Grant



sec_r0

Authorization Code Grant

* This is the most secure grant over the other grants.
* When user authorizes the application in step 5 the redirect happenes to the application with authz code in URI, shown in step 6

Request in step 5, look like this

```
https://authz-server/oauth/authorize
?client_id=123456
&response_type=code
&state=654321
&redirect_uri=https%3A%2F%2Fapp.com%2Fauth
&scope=photos
```

- `client_id=123456` → your app identifier
- `response_type=code` → Authorization code grant is used
- `state=654321` → this is preset data passed to client app
- `redirect_uri=https%3A%2F%2Fapp.com%2Fauth` → Auth code will be returned to this URL
- `scope=photos` → optional scope where this code will be valid

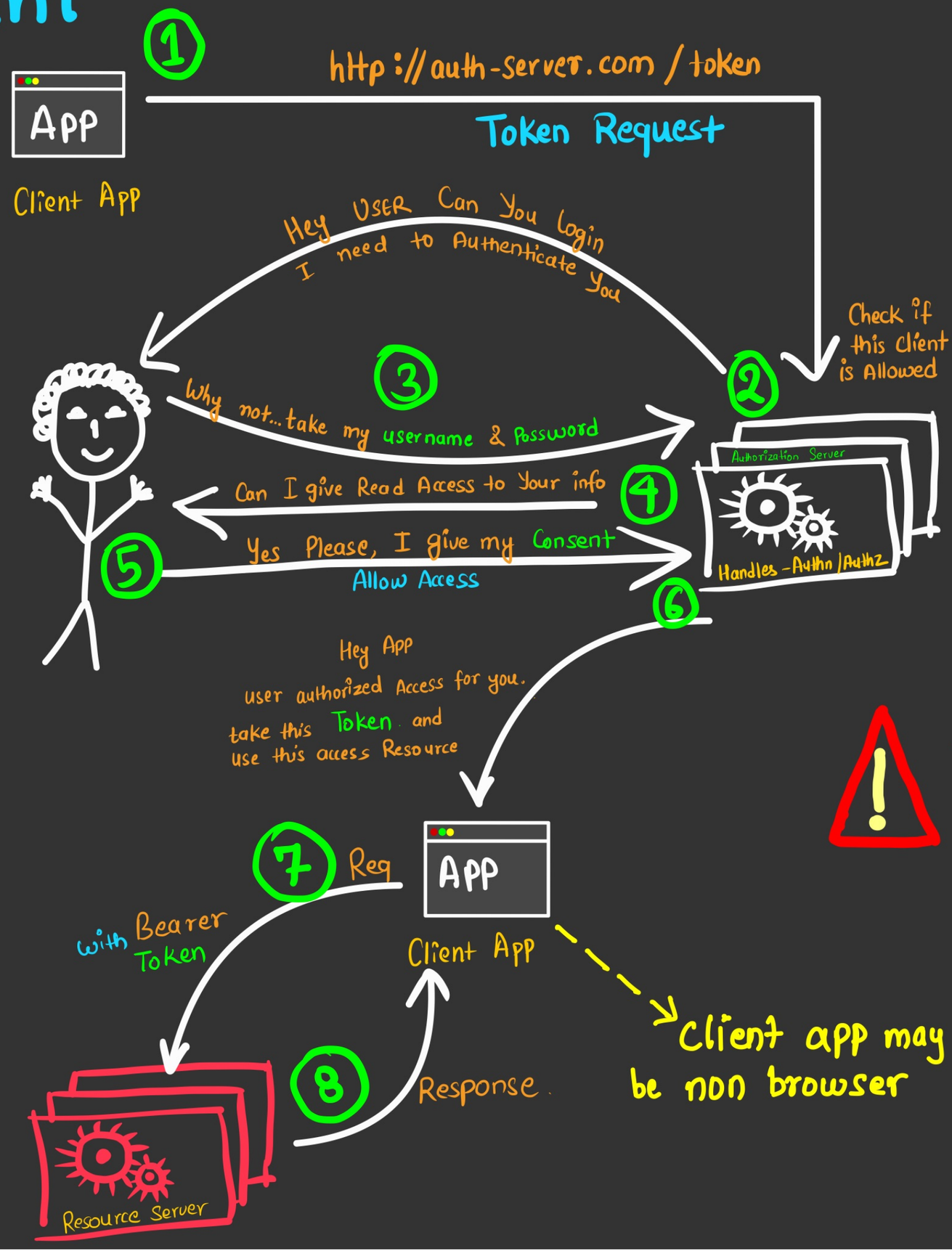
** User Authentication Credentials are passed in body and should be encrypted.

* exchange for the code for access token in step 7 is authenticated with client_secret
* Access token can be encrypted using JWE which will make this grant more secure

** I will discuss about PKCE in OAuthZine in detail.

sec_r0

Implicit Grant



Implicit Grant ⚠ This is not recommended

* This grant is similar to Auth Code grant except that after step 6, client app directly gets the access token.

* This is less secure because client credentials are not validated.

If you remember in Auth-Code-grant in step 6, client Credentials are validated.

⇓

which implies when App is running in browser using Implicit grant the access token will be stored in browser, making access_token available to web App attacks like XSS etc

* This case never comes with Auth Code, as access_token has to be requested each time and on each request to access token Client Credentials are validated.

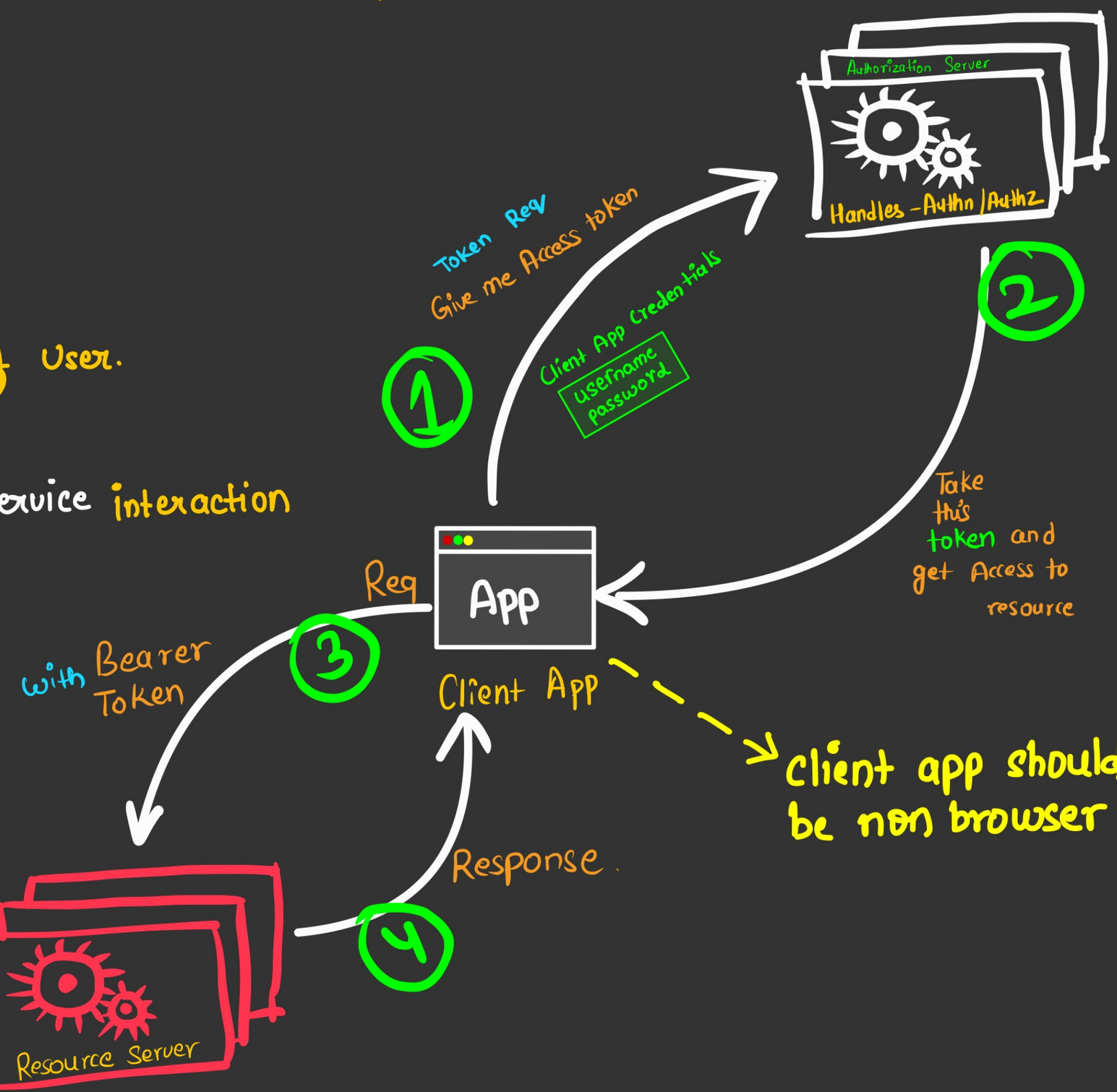
sec_r0

Client Credentials Grant

* used when client is acting on its own behalf (when client is resource server)
* The flow is minimilistic of all, and used when client app is intrested in fetching its own protected resource.

* There is no involvement of User.

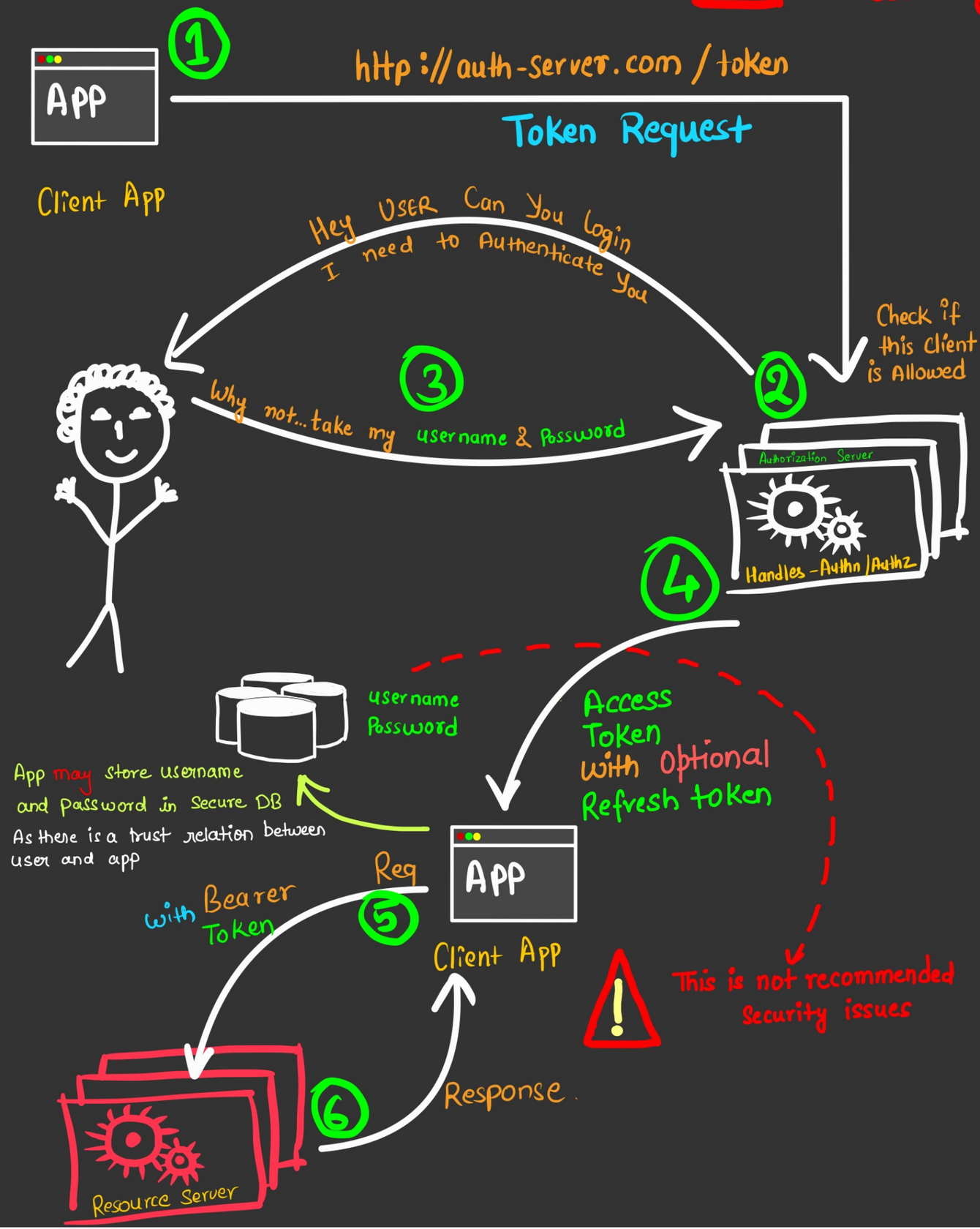
And this makes it suitable for service-to-service interaction



sec_r0

Resource Owner Password Credential Grant

This is not recommended
Security issues



sec_r0

Resource Owner Password Credential Grant

* This flow works well when resource owner  have trust relation with 

→ eg client is an operating system could be trusted. Similar to the way we use Mac's Keychain to store our creds.

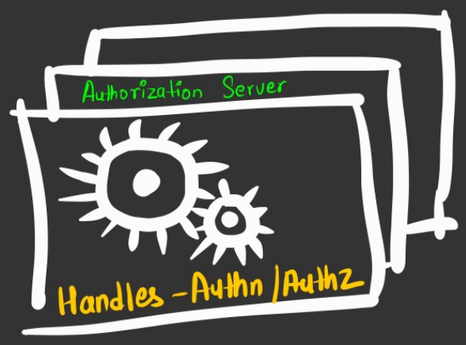
* This grant was proposed to migrate app from HTTP basic auth to OAuth by Converting **Stored Credentials** to access_token.

* Exposing User Credentials to 3rd party.

* Also in this grant user have no control to the Authorization process as below step is missing.

Yes Please, I give my Consent

Allow Access

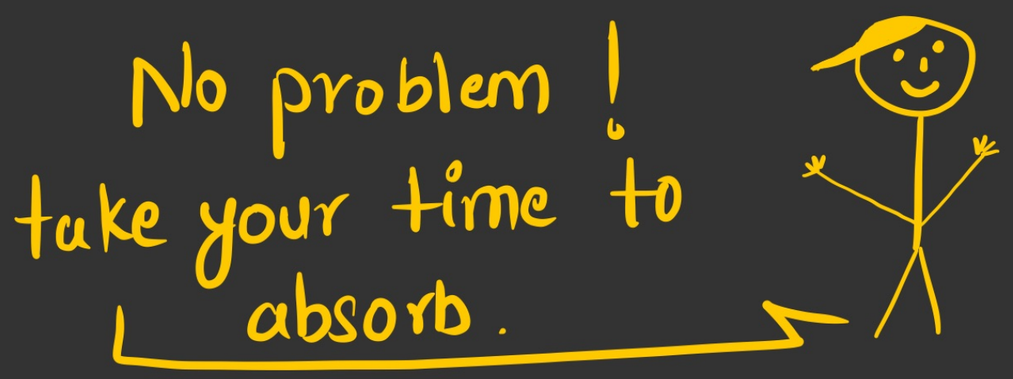


* Once user enter Credentials BOOM entire Credentials Control is with the App.

No No No !!

If you have done this, quickly reset your password.

sec_r0

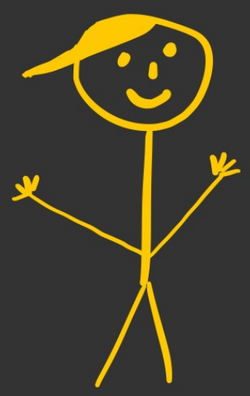


I have one last topic to discuss and that is OIDC

The same OIDC that you mentioned while discussing the difference between AuthN and AuthZ ?

You know,
You are smart

OpenID Connect



Since Now you have seen OAuth
You know that Authorization
servers have capability to
validate user's credentials, right?

Yes, Exactly I
was about to
ask.....
So, Can AuthZ servers
become AuthN servers
too...!

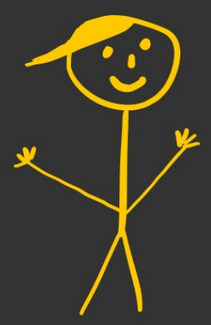
Bing, bing, Bingo
Thats what OIDC helps in.

OIDC extends the capability of Authorization
server to share user identity and behaves
as Identity provider ⇒ Behaving as
Authentication Server

sec_r0

OpenID Connect

If you remove this part. This identity provider will no longer be identity provider. Rather It will become OAuth server.



1. I want to access this Application Resource *

2. Scope = OpenId — Redirect (302) the user to Authorization Server with client id.

3. I need to Know who are you
   Sure, I am Sec_r0 and these are my creds

4. username, Password

5. Verify user Credentials

6. Credentials OK

7. The user gets redirected to application with authorization code

8. Exchange the authorization code to get an access token

9. access_token + id_token as JWT

10. Gimme User info take this access_token + id_token

11. take this user info as JWT
    Signed by Google

12. So You are Sec_r0 ..... Welcome. *

App

Endpoint /authorization

Endpoint /token

Endpoint /userinfo

Server

Track User authentication *
- who is already logged in
- when they logged in

All in here →

Google Identity Provider **

Note

* id_token Returned by Google Idp in step 9. Contains the authentication status and info

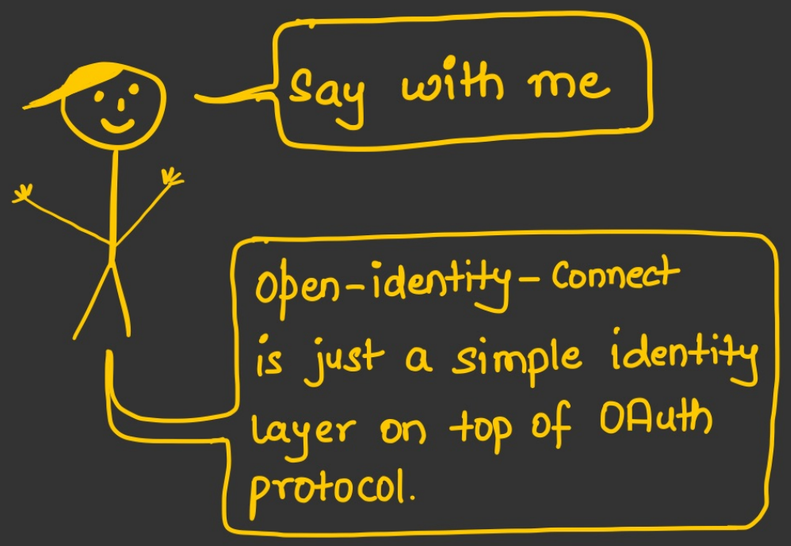
The flow in white color is part of OAuth Already. [Here :- Auth Code flow]

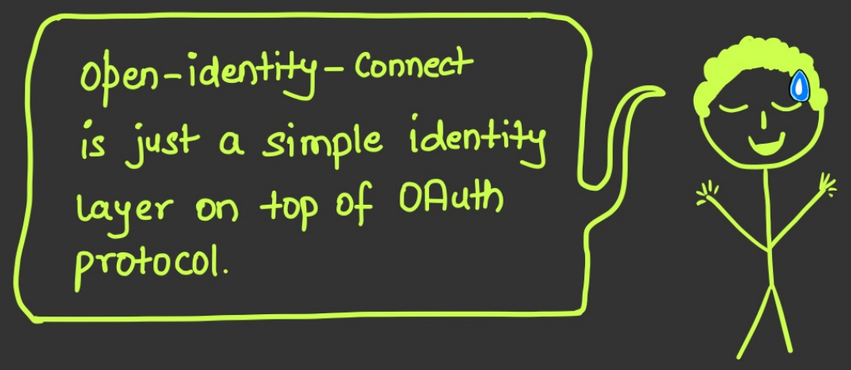
The flow in baby pink color is addition of OIDC flow on top of OAuth.

sec_r0

** Google Identity Provider is a reference here not endorsement

OpenID Connect





* This works/is defined to work with web based, mobile and Javascript clients provided you use secure grant types from OAuth.

I never imagined OIDC is simple if you know OAuth.

sec_r0

A big thanks to awesome reviewers

Content Reviewed, Examined & Corrected By

← Christina

theIAMNinja

Identity And Access Management

Thanks to you Christina

@apisecurityio ← Must checkout security newsletter.

@DSotnikov ← Maintainer of this project

@nirali0 ←

@susam ←

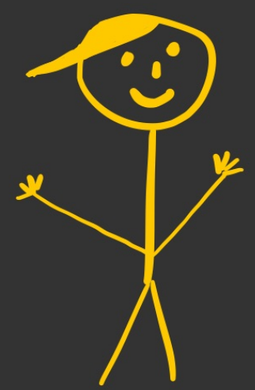
@KhajaSubhani ←

Sneha Anand ←

4 great human beings, who believed and Supported in Security Zines initiative.

I hope you enjoyed !

Thanks for Reading



sec_r0

Read More Zines @

securityzines.com