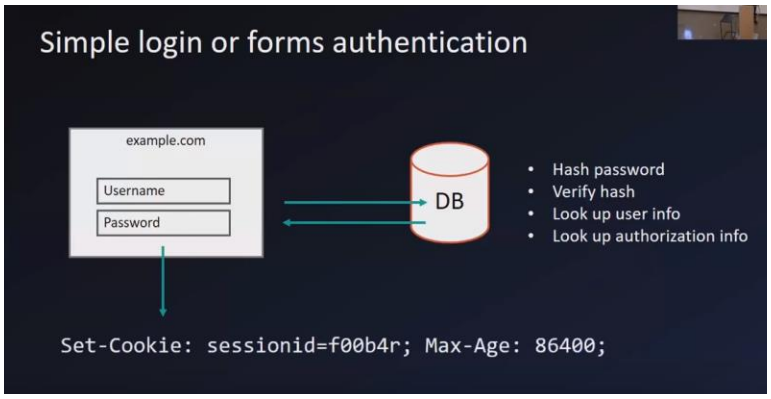
OAuth 2.0 & OpenID Connect (OIDC)

Contents

History of Authentication & Authorization

Simple Login/Form Authentication



```
Set-Cookie: sessionid=f00b4r; Max-Age: 86400;
```

Here typically the user enters the username & password. This in-turn will hit the database, verifies the username & password (may be using password hash, encrypted password to avoid storing the password as plain text) and performs the authorization.

This principle was used across the industry, but it had its own disadvantages/fallback

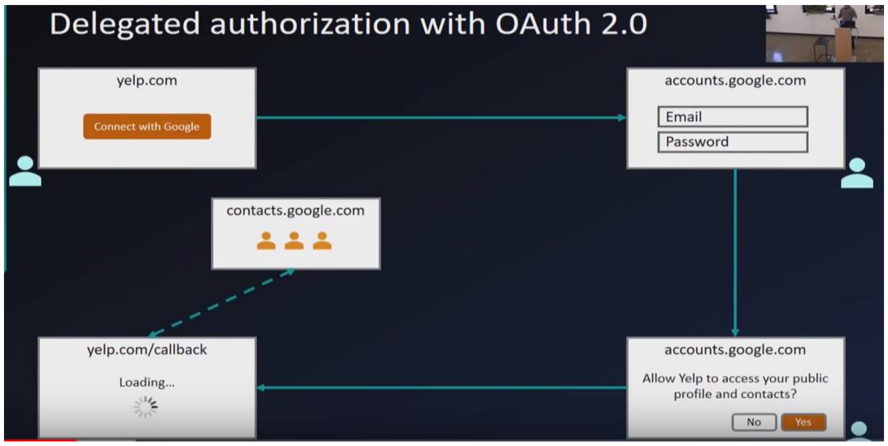
- ❖ **Security Issue** – Developers needs to secure the information, encrypts the passwords, resolve penetration/vulnerability testing issues.
- ❖ **Maintenance** – Developers needs to maintain this information securely as the user information is stored in DB

Use Cases for OAuth:

- ✓ **Single Sign On (SSO)**
- ✓ **Delegated Authorization**, i.e., when we install any app & sign in using our Google/Facebook account and it asks permission such as it can access my contacts, my profile, read my messages etc.,

Use Case: OAuth 2.0 - Delegated Authorization:

Here let's consider a website, yelp.com is trying to authorize the user using user's Google Account. Once Authorized, they seek user's permission to get access to their contacts. Once user grants the permission, yelp will be able to get access to user's contacts list.



Steps Involved:

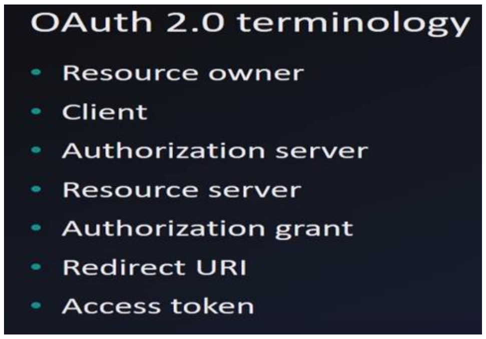
- ✓ User opens yelp.com & sees 'Connect with Google' Option. With this, User can authorize himself using his Google Account. So, User clicks on 'Connect with Google' Option.

- ✓ Once clicked, user is routed to Google Website (accounts.google.com) where he can authenticate himself by providing his Google Username & Password. The important thing to note here is, Google is performing the authentication (validating the username & password) and not Yelp.

- ✓ On Successful authentication, Google prompts a message seeking permission for Yelp to access the user's Profile & Contacts.

- ✓ Assuming the user clicks on 'Yes', user will be redirected to the Callback endpoint of Yelp.com. Since the user has clicked 'Yes', Yelp will now have access to connect with Google to get your Profile & Contacts by making a REST call to Google's Contact API [contacts.google.com]

OAuth 2.0 Terminologies:



OAuth 2.0 terminology

- Resource owner
- Client
- Authorization server
- Resource server
- Authorization grant
- Redirect URI
- Access token

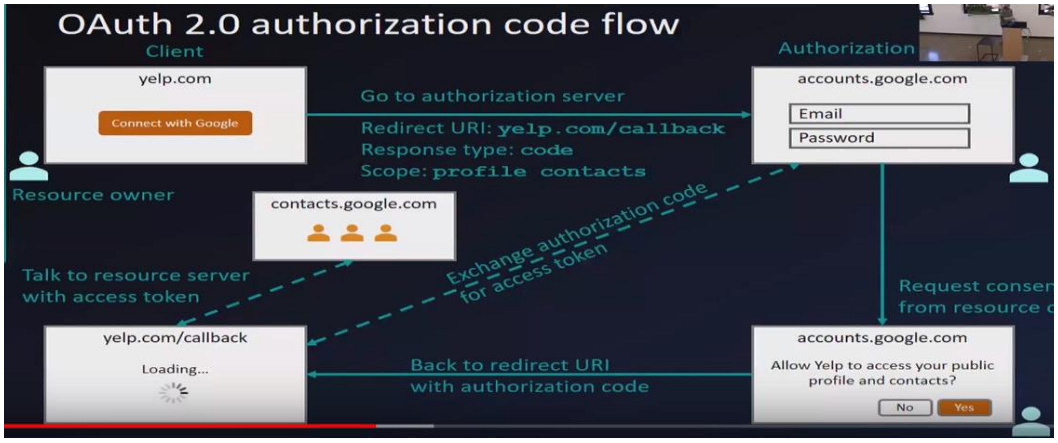
- ✓ **Resource Owner** – the one who owns the data i.e., the Users (either you or me)
- ✓ **Client** – the one who seeks to perform the authorization. Example: Yelp.com, from the above use case.
- ✓ **Authorization Server** – the one who perform the authorization. Example: Okta or Google.com from the above use case
- ✓ **Resource Server** – the one who have the data which the Client is requesting. Example: contacts.google.com from the above use case
- ✓ **Authorization grant** – the grants provided by the users. Example: User clicks on 'Yes' for accessing his Profile & Contacts.
- ✓ **Redirect/Callback URI** – It is the call back URL. From the Authorization server, when the user clicks on Yes, it should know where to redirect the user to. Example: After the user clicks on Yes, user is redirected to **Yelp.com/callback**
- ✓ **Access Token** – It is the token generated by the Authorization Server (Google.com) & given to the Client (Yelp.com) to perform the Authorization while calling the Resource Server (contacts.google.com).

Note: In Some cases, Authorization Server can be same as the Resource Server (if both are deployed on to the same instance).

OAuth2.0 Flow Types:

- ✓ Authorization Code Flow
- ✓ Implicit Flow

OAuth 2.0 – Authorization Code Flow:


OAuth 2.0 authorization code flow
Client
yelp.com
Connect with Google
Authorization
accounts.google.com
Email
Password
Go to authorization server
Redirect URI: yelp.com/callback
Response type: code
Scope: profile contacts
Resource owner
contacts.google.com
Exchange authorization code for access token
Talk to resource server with access token
Request consent from resource o
yelp.com/callback
Loading...
Back to redirect URI with authorization code
accounts.google.com
Allow Yelp to access your public profile and contacts?
No
Yes
[Image 1]

User visits Yelp.com & Clicks on 'Connect with Google' Option.

- ✓ **Step 1:** Yelp.com (Client) Sends a request to the Authorization Server with Redirect URI, Response type & Scope attributes.
  - **Client** sends the Redirect URI initially itself, because (once the user clicks on 'Yes' in the consecutive steps) Client needs to redirect the user to call back URI.
  - **Response Type**: Code, referring to Authorization Code Flow.
  - **Scope:** Limiting the scope of the requests to **profile** & **contacts**. So, Authorization server will generate the Authorization Code only for Profile & Contacts & not for read/delete emails

- ✓ **Step 2:** User is redirected to Authorization Server (Google.com) & user enters the credentials and performs authentication. Once Authenticated, it shows a pop up stating that the Client (Yelp.com) requires permission to access Profile & Contacts.

  Note: Authentication (entering the username & password and Validation) happens at Authorization Server & not at Client Side.

- ✓ **Step 3:** (Assuming the user clicks on 'Yes'). Authorization Server redirects the user to the Redirect URI sent by the Client in Step 1, i.e., (Yelp.com/callback) with Authorization Code.

- ✓ **Step 4:** (Assuming successfully redirected to Client's callback URI). Client makes another REST call to Authorization Server with Authorization Code and gets back an Access Token i.e., Exchanging Authorization Code for Access Token.

  Note: Authorization server will validate the Authorization Code, like whether it is valid & proper and then returns the Access token to the Client.

- ✓ **Step 5:** Client makes a REST call to Resource Server (contacts.google.com) with Access Token in the Request Header. Resource Server will validate the access token & then provides the response (sends back the User's Profile & Contacts only).

  Note: Resource server will provide access only to contacts.google.com & not for any other API's like mails.google.com (for reading the user's emails) etc., since we have already sent the Scope values as profile contacts.

Why the additional step of getting the Authorization Code & again exchanging the Authorization Code for an Access Token with Authorization Server?

To explain this, we need to understand the concept of Front Channel & Back Channel.

- **Front Channel (Less Secure):** Here the requests happens via **browser** (Client call). Since the browser has many open loops, chances are there to steal the data of the user.

- **Back Channel (Highly Secure):** Here the requests happens via **server** (Backend calls, from server to server [API calls]). So here the chances are less or zero, & the data exchanged or shared is highly secure.

With reference to above Image 1, the solid line refers to Open Channel i.e., from Client to Authorization Server & redirecting back to the callback URI of Client happens via Browser (simply browser redirects only) & the dotted line refers to Back Channel (API calls from Backend Server). So, here the Authorization Code is also shared via browser (may be as query param). This has potential threat & any ethical hackers with malicious tools installed on the browser can easily take/steal the Authorization Code.

This is the main reason that an extra call is again made from the Client's (Backend Server) to the Authorization Server's (Backend Server) to exchange an Access Token for an Authorization Code.

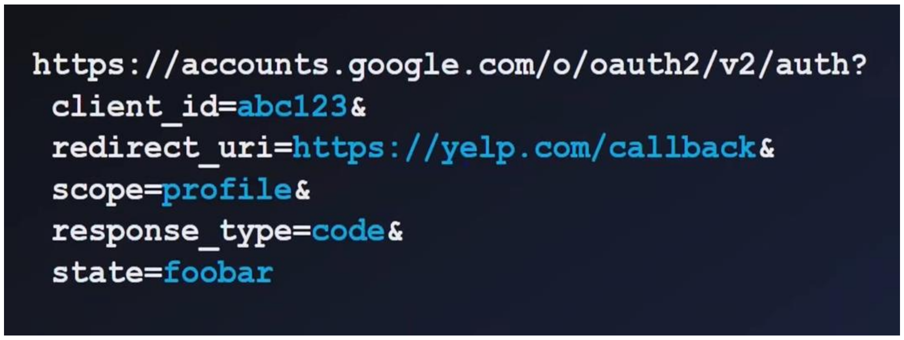
Also, to make this process very secure, Client has a secret key called Client Secret Key which only the Client knows it & will be used during the Exchange Process with the Authorization Server.

How Authorization Server knows about this Client & how does it know that Client (Yelp.com) is calling the Authorization Server?

So Initially Clients (yelp.com) needs to do some configuration with the Authorization Server. For instance, for Google as an Authorization Server, you need to create an Authorization server for Google using Google API Console where Google will provide you two things.

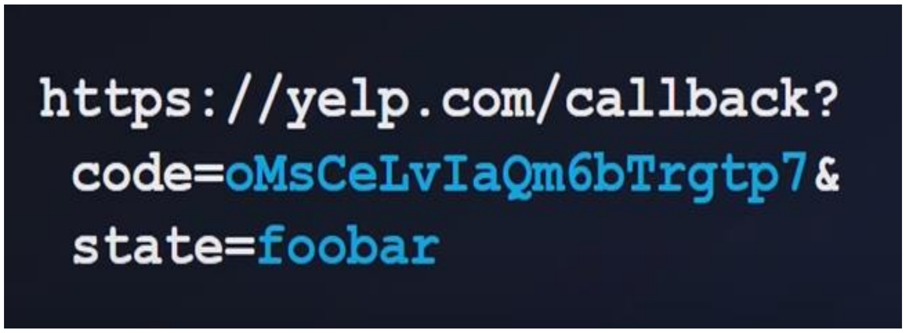
- Client ID
- Client Secret

Sample Request from Client to Authorization Server:



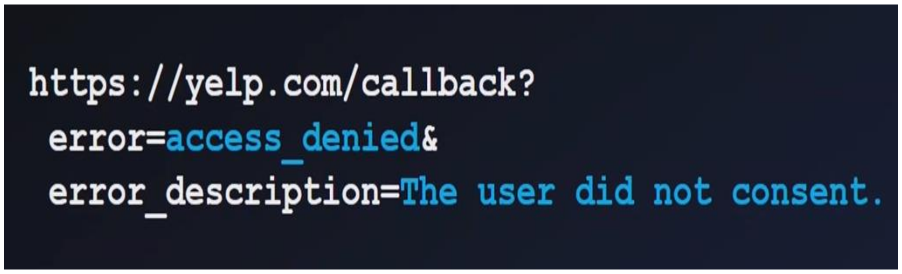
```
https://accounts.google.com/o/oauth2/v2/auth?
 client_id=abc123&
 redirect_uri=https://yelp.com/callback&
 scope=profile&
 response_type=code&
 state=foobar
```

Sample Success Response from Authorization to Client:



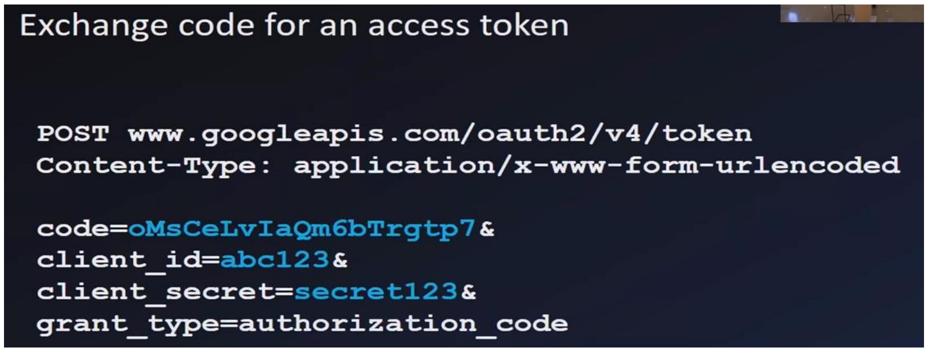
```
https://yelp.com/callback?
 code=oMsCeLvIaQm6bTrgtp7&
 state=foobar
```

Sample Failure Response from Authorization Server to Client:



```
https://yelp.com/callback?
 error=access_denied&
 error_description=The user did not consent.
```

Sample Request for exchanging Authorization Code for Access Token



Exchange code for an access token

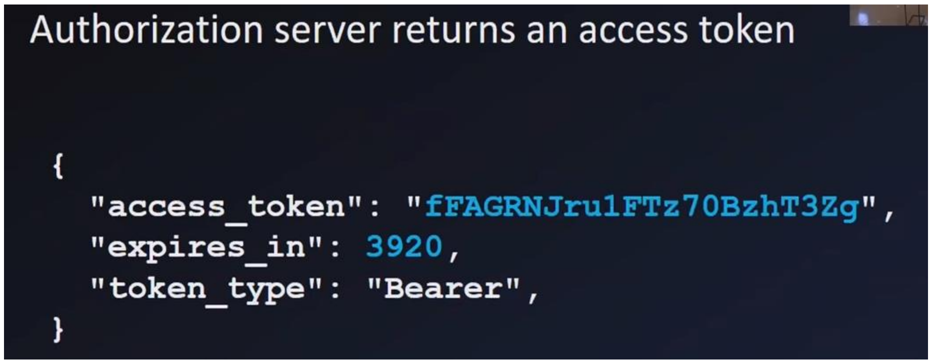
```
POST www.googleapis.com/oauth2/v4/token
Content-Type: application/x-www-form-urlencoded

code=oMsCeLvIaQm6bTrgtp7&
client_id=abc123&
client_secret=secret123&
grant_type=authorization_code
```

From the above response,

- **Code** refers to Authorization Code received from Authorization Server.
- **Client ID** refers to unique ID of Client which was created while creating the Authorization Server.
- **Client Secret** refers to the Secret Code of Client which was created while creating the Authorization Server.
- **Grant_type** refers to type of grant i.e., we are asking the Authorization server to grant the Access token in exchange of Authorization Code.

Sample Success Response from Authorization Server with Access Token to Client:



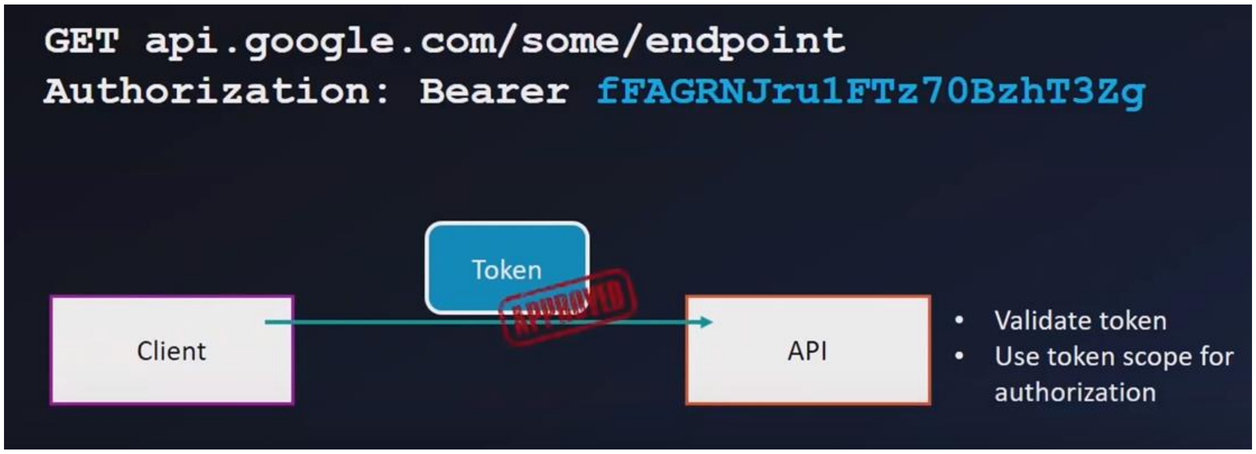
Authorization server returns an access token

```
{
  "access_token": "fFAGRNJru1FTz70BzhT3Zg",
  "expires_in": 3920,
  "token_type": "Bearer",
}
```

From the above response,

- **Access_token** refers to Access Token returned from Authorization Server
- **Expires_in** refers to expiration time limit
- **Token_type** refers as Bearer Token

Sample for making an API call to Resource Server with Access Token (Bearer Token) in Request Header



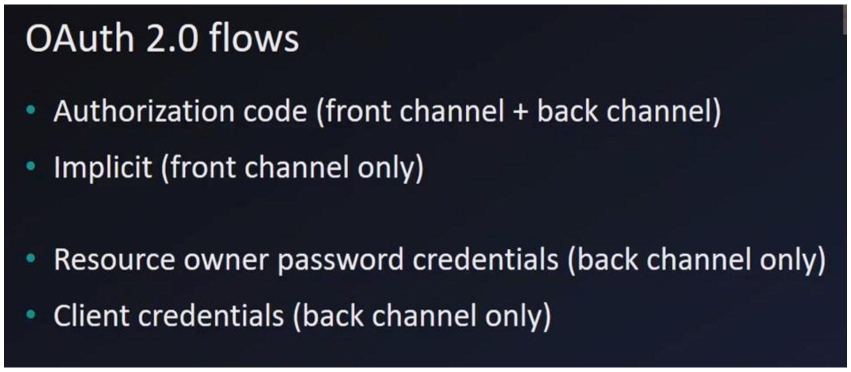
```
GET api.google.com/some/endpoint
Authorization: Bearer fFAGRNJru1FTz70BzhT3Zg
```

- Validate token
- Use token scope for authorization

So here,

- whenever the Client makes a request to API's in Resource Server, it attaches the Access Token or Bearer Token in the Request Header.
- The Resource Server ensures that the Bearer Token is Valid (Proper as well as not expired) & allows the Client to access the API.
- It also makes sure that the Client is accessing only the required API (whatever mentioned in the scope). i.e., for instance, it accesses only the read contacts API & not delete Contacts API.

Other OAuth2 Flows:

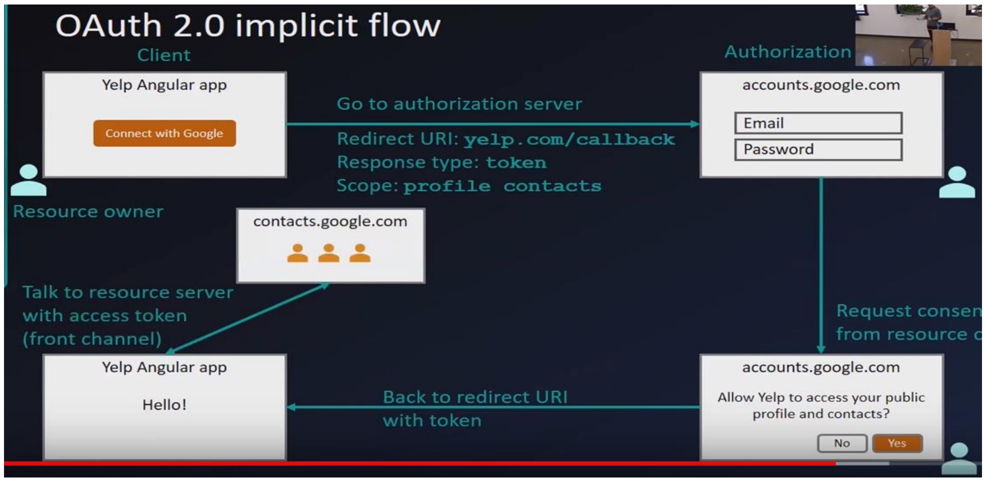


OAuth 2.0 flows

- Authorization code (front channel + back channel)
- Implicit (front channel only)
- Resource owner password credentials (back channel only)
- Client credentials (back channel only)

- Authorization Code Flow: Explained above.
  Use Case: Angular/React JS as Front End and Spring Boot as Back End
- Implicit Flow: Only Front Channel
  Use Case: Pure Angular/React JS with no backend
- Resource owner password credentials: Only Back Channel.
  Use Case: Pure Spring Boot App with no Front End.
- Client Credentials: Only Back Channel. Use Case: Machine to Machine Interaction.

Implicit Flow:



This is same as Authorization Code Flow, except for the below.

- Here, Client sends the response type as '**token'** instead of '**code'**. This indicates the Authorization Server to return the Access Token directly instead of sending back the Authorization Code & exchanging back for an access token.
- All the calls are made through front channel i.e., through browser.
- This flow is applicable only for an application which is built using Pure JavaScript/Angular/React JS without any Backend Server for processing the data & rendering the UI.

Use Cases of OAuth before 2014:

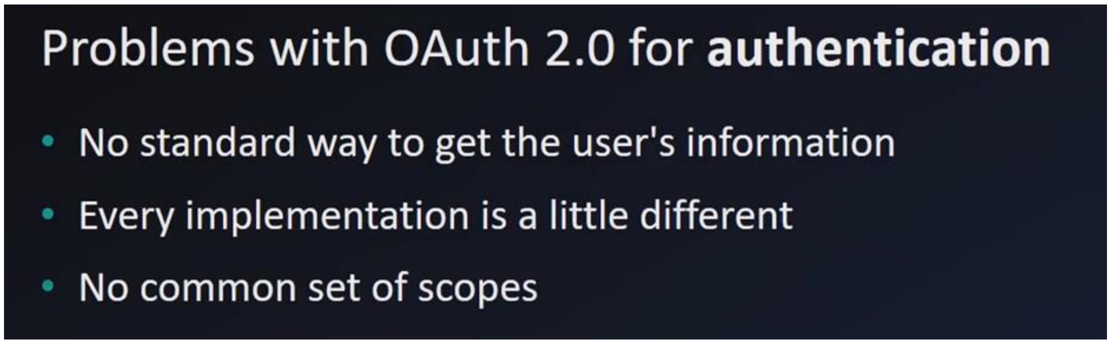
Some of the use cases of OAuth before 2014 are

- Simple Login - Authentication
- Single Sign On (SSO) – Authentication
- Delegated Authorization - Authorization
- Mobile App Login - Authentication

Before 2014, OAuth was also used for Authentication Purpose. But typically, OAuth was not designed for Authentication purpose rather it is for Authorization Purpose. It focuses mainly on permissions, accessing/granting access tokens, specifying the scope of token's for API's etc.,

So, OAuth was designed for Authorization & not for Authentication.

Reason why OAuth should not be used for Authentication



Problems with OAuth 2.0 for **authentication**

- No standard way to get the user's information
- Every implementation is a little different
- No common set of scopes

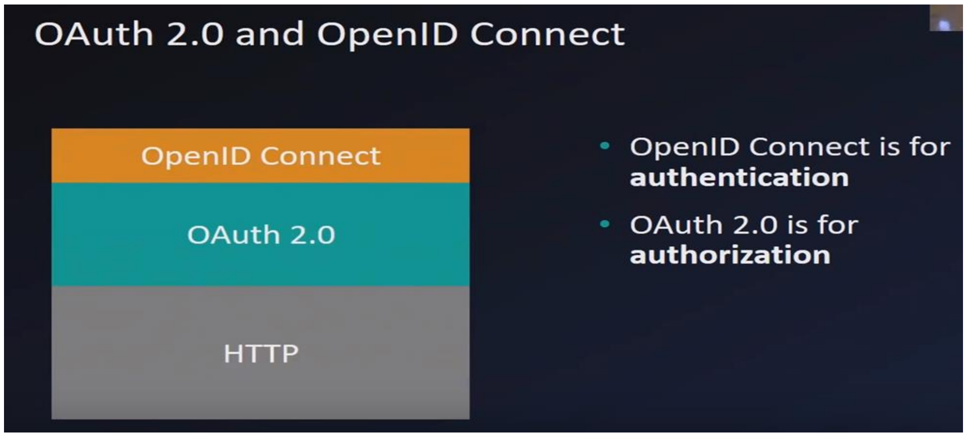
Main reason being

- For instance, if a user logs into Website using OAuth, it doesn't satisfy the website's criteria for getting the user's information. May be the website may require additional useful information about the user which may not be available from the data returned by the Authorization Server.

  So, there is no standard way of getting the user's information.

OpenID Connect

How OpenID Connect solved the Authentication Problem for OAuth2

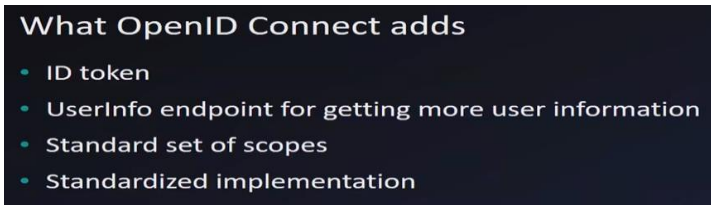


OAuth 2.0 and OpenID Connect

- OpenID Connect is for **authentication**
- OAuth 2.0 is for **authorization**

Without changing the design of OAuth2, a standard layer called OpenID Connect was added on top of OAuth2 for Authentication purpose.

Essentials of OpenID Connect for Authentication:

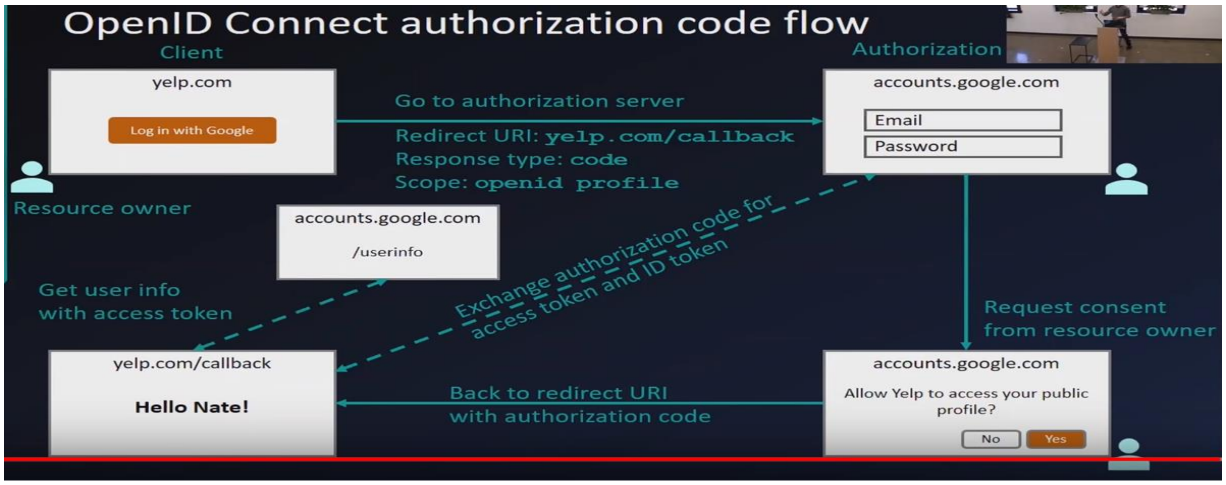


What OpenID Connect adds

- ID token
- UserInfo endpoint for getting more user information
- Standard set of scopes
- Standardized implementation

- With OpenID Connect, the Client request the Authorization Server for ID token i.e., JWT Token (Json Web Token) in the scope parameter which contains basic user information.
- If the information required by your app is not available from the basic user information, the Client can make a call to /userInfo Endpoint.
  The /userInfo is a standard API which is available at all Authorization Server & would understand what the Client is referring to.
- Open ID Connect has a standard set of scopes for getting the necessary information

OpenID Connect Authorization Code Flow:

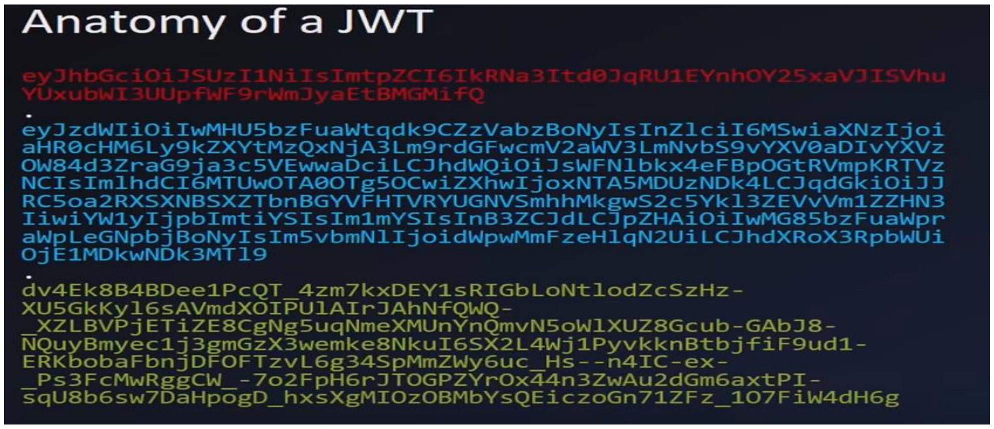

OpenID Connect authorization code flow
Client
Authorization
yelp.com
Log in with Google
Go to authorization server
Redirect URI: yelp.com/callback
Response type: code
Scope: openid profile
accounts.google.com
Email
Password
Resource owner
accounts.google.com
/userinfo
Exchange authorization code for access token and ID token
Get user info with access token
Request consent from resource owner
yelp.com/callback
Hello Nate!
Back to redirect URI with authorization code
accounts.google.com
Allow Yelp to access your public profile?
No
Yes

The differences that the OpenID Connect adds to the OAuth2 is that,

- In the first call from Client to Authorization Server, in the scope parameter, Client will be sending openid as a parameter along with the other required scopes.
- After redirecting to the callback URI, the client exchanges the Authorization code for access token as well as for ID token (JWT Token)
- So, with the ID token (JWT Token), Client can grasp the basic user information such as first name, last name, email address etc.,
  Note: If you see the above image, after the exchange process – the Client is now able to get the user information – first name & displaying as 'Hello **Nate**!'.

  If more user information is required, Client can make a request to /userInfo endpoint in Authorization server by passing the access token.

Anatomy of ID Token (JWT Token)

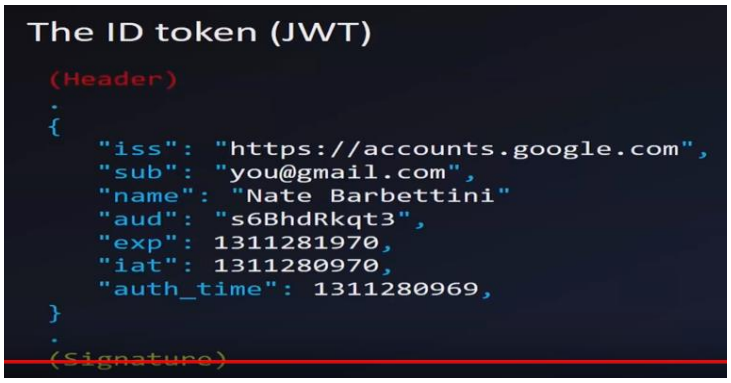


Anatomy of a JWT

```
eyJhbGciOiJSUzI1NiIsImtpZCI6IkRNa3Itd0JqRU1EYnhOY25xaVJISVhu
YUxubWI3UUpfWF9rWmJyaEtBMGMifQ
.
eyJzdWIiOiIwMHU5bzFuaWtqdk9CZzVabzBoNyIsInZlciI6MSwiaXNzIjoi
aHR0cHM6Ly9kZXYtMzQxNjA3Lm9rdGFwcmV2aWV3LmNvbS9vYXV0aDIvYXVz
OW84d3ZraG9ja3c5VEwwaDciLCJhdWQiOiJsWFNlbkx4eFBpOGtRVmpKRTVz
NCIsImlhdCI6MTUwOTA0OTg5OCwiZXhwIjoxNTA5MDUzNDk4LCJqdGkiOiJJ
RC5oa2RXSXNBSXZTbnBGYVFHTVRYUGNVSmhhMkgwS2c5Ykl3ZEVvVm1ZZHN3
IiwiYW1yIjpbImtiYSIsIm1mYSIsInB3ZCJdLCJpZHAiOiIwMG85bzFuaWpr
aWpLeGNpbjBoNyIsIm5vbmNlIjoidWpwMmFzeHlqN2UiLCJhdXRoX3RpbWUi
OjE1MDkwNDk3MTl9
.
dv4Ek8B4BDee1PcQT_4zm7kxDEY1sRIGbLoNtlodZcSzHz-
XU5GkKyl6sAVmdXOIPUlAIrJAhNfQWQ-
_XZLBVPjETiZE8CgNg5uqNmeXMUnYnQmvN5oWlXUZ8Gcub-GAbJ8-
NQuyBmyec1j3gmGzX3wemke8NkuI6SX2L4Wj1PyvkknBtbjfiF9ud1-
ERKbobaFbnjDFOFTzvL6g34SpMmZWy6uc_Hs--n4IC-ex-
_Ps3FcMwRggCW_-7o2FpH6rJTOGPZYrOx44n3ZwAu2dGm6axtPI-
sqU8b6sw7DaHpogD_hxsXgMIOzOBMbYsQEiczoGn71ZFz_1O7FiW4dH6g
```

- Words in Red refers to Header (contains the payload information)
- Words in Blue refers to Claims (Claims will be used by the app. It first decodes it & retrieves the basic user information)
- Words in Yellow refers to Signature (optional additional step may be performed by your app to make sure that the token is not forged or changed)

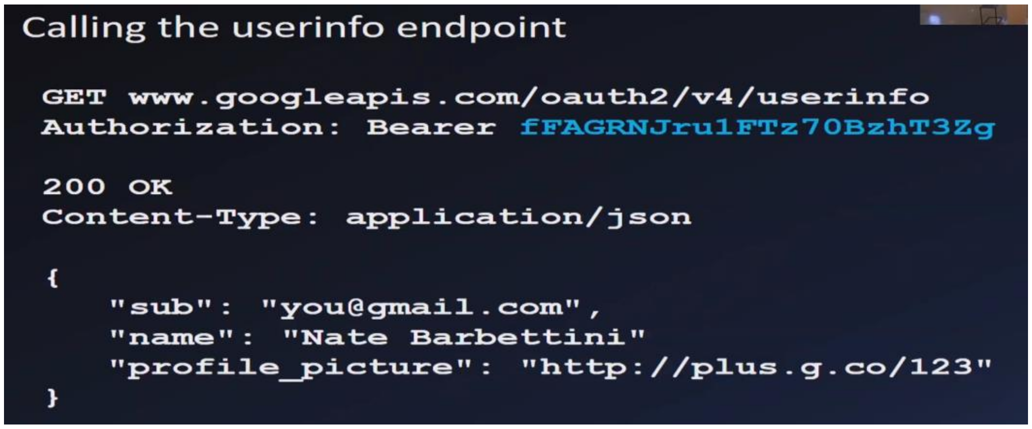
Decoded ID Token (JWT Token):



The ID token (JWT)

```
(Header)
.
{
    "iss": "https://accounts.google.com",
    "sub": "you@gmail.com",
    "name": "Nate Barbettini"
    "aud": "s6BhdRkqt3",
    "exp": 1311281970,
    "iat": 1311280970,
    "auth_time": 1311280969,
}
.
(Signature)
```

Getting Additional Information by calling the /userInfo Endpoint:



Calling the userinfo endpoint

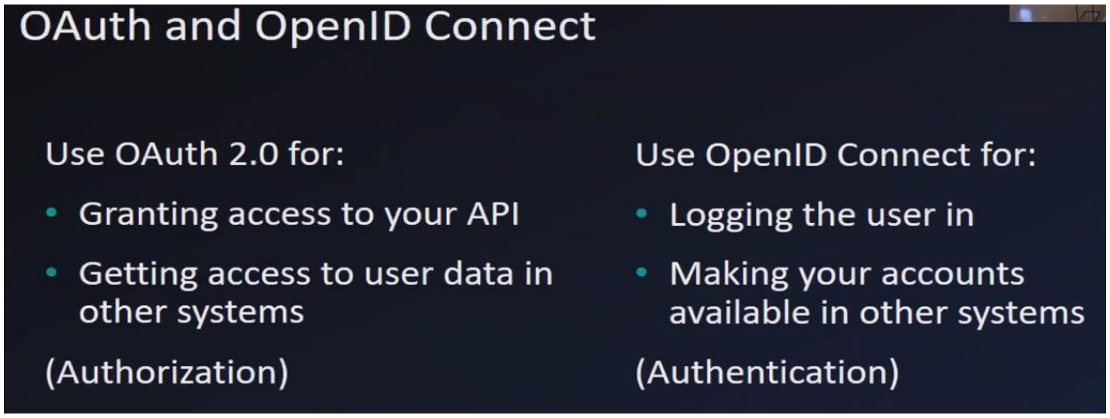
```
GET www.googleapis.com/oauth2/v4/userinfo
Authorization: Bearer fFAGRNJru1FTz70BzhT3Zg

200 OK
Content-Type: application/json

{
    "sub": "you@gmail.com",
    "name": "Nate Barbettini"
    "profile_picture": "http://plus.g.co/123"
}
```

So, here the Client makes a call to /userInfo endpoint by attaching the Bearer Token (Access Token) & gets additional information about the user such as Profile Picture etc.,

So, finally the difference b/w OAuth2 & OpenID Connect is,



OAuth and OpenID Connect

Use OAuth 2.0 for:

- Granting access to your API
- Getting access to user data in other systems

(Authorization)

Use OpenID Connect for:

- Logging the user in
- Making your accounts available in other systems

(Authentication)

References:

https://speakerdeck.com/nbarbettini/oauth-and-openid-connect-in-plain-english

https://www.youtube.com/watch?v=996OiexHze0