THIS IS JUST AN OF EXAMPLE RESEARCH

OAuth 2.0 & OpenID Connect

https://aaronparecki.com/oauth-2-simplified/
Discussion on why Implicit Flow and Resource Owner Password Credentials Grant are bad:
https://www.ory.sh/oauth2-for-mobile-app-spa-browser

Roller

Klient - app som trenger tilgang til *bruker* sine ressurser.
Ressursserver - serveren med ressursene til *brukeren*. Gir bare tilgang til ressurser hvis det presenteres en access token.
Autoriseringsserver - server som sjekker om *klient* har lov fra *bruker* til å motta access token og sender en access token til *klient* hvis *klient* har tillatelse. Kan f.eks. gjøres ved å vise en side der *bruker* kan gi sin tillatelse/godkjenning(typ sånn som google gjør ved innlogging fra ny enhet). Kan ofte være lagt på samme server som *ressursserver.*
Bruker - "Eieren" av ressursene(i dette tilfellet en ansatt på sykehus). *Bruker* må gi sin tillatelse til *Klient* og bruke sitt passord og brukernavn for tilgang til en access token.
Ressurser - *Klient* har tilgang til f.eks. Pasientlister, rutiner, behandlingsmåte etc.

Klient-ID og hemmelighet

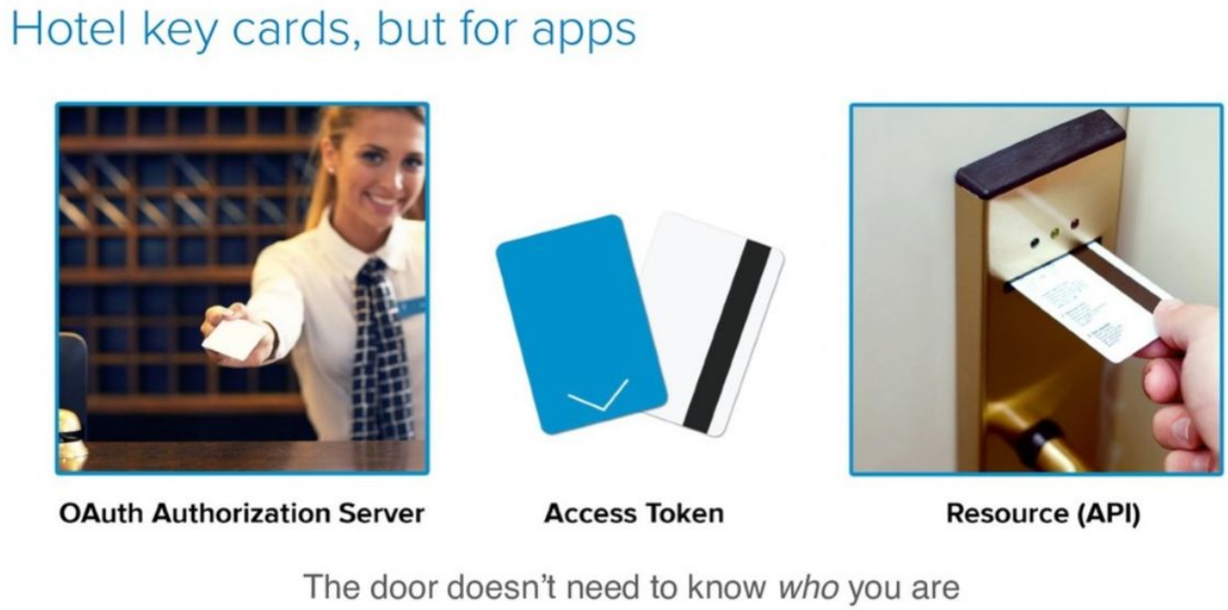
Når vi registrerer appen vår med *autoriseringsserveren* ("the service") får vi en klient-ID. Klient-IDen er offentlig og brukes til å bygge login-URLer. Man får vanligvis en klient-hemmelighet, men denne kan visst ikke holdes hemmelig på native-apps[1], og da bruker man den ikke.

---

[1] "OAuth 2 Simplified • Aaron Parecki." https://aaronparecki.com/oauth-2-simplified/. Accessed 24 Jan. 2019.

Authorization vs. Authentication

What is authorization?



Hotel key cards, but for apps

OAuth Authorization Server

Access Token

Resource (API)

The door doesn't need to know *who* you are

The card authorizes your access to the resource (your hotel room)

Authorization

Authorization grant - representerer *bruker* sin autorisasjon om at *klient* har tilgang til *ressurser.* Finnes 4 typer:

- Authorization Code
    - Brukes av selve appen på vegne av seg selv (ikke brukeren) til å få oppdateringer og generelle stats om brukerne av appen
- Implicit
- Resource owner password credentials
    - *Bruker* sitt passord og brukernavn brukes som *authorization grant* av *klient* og sendes til *autoriseringsserver* for tilgang til en *access token*. Det betyr at *klient* har tilgang til *bruker* sitt passord og brukernavn.
- Client credentials

Access token -

Access tokens are used as bearer tokens. A bearer token means that the bearer can access authorized resources without further identification. Because of this, it's important that bearer tokens are protected. If I can somehow get ahold of and "bear" your access token, I can masquerade as you.

These tokens usually have a short lifespan (dictated by its expiration) for improved security. That is, when the access token expires, the user must authenticate again to get a new access token limiting the exposure of the fact that it’s a bearer token.[2]

Refresh token

Refresh tokens are used to obtain new access tokens. Typically, refresh tokens will be long-lived while access tokens are short-lived. This allows for long-lived sessions that can be killed if necessary.[3]

- A special token used to get new access tokens
- Requested along with the access token in the initial step
- Usually requires scope: offline_access
- Usually not issued to Javascript clients
  - Due to security concerns and because we don’t expect the Javascript apps to need access after the user stops using them actively

Exchange refresh token for Access Token:

```
POST https://authorization-server.com/token
  grant_type=refresh_token&
  refresh_token=REFRESH_TOKEN&
  redirect_uri=REDIRECT_URI&
  client_id=CLIENT_ID&
  client_secret=CLIENT_SECRET
```

Response includes new access token (and possibly also a new refresh token, not defined by the spec) Assume that if you do get a new refresh token, use that.

```
{
  "access_token": "RsT5OjbzRn430zqMLgV3Ia",
  "expires_in": 3600,
  "refresh_token": "64d049f8b21191e12522d5d96d5641af5e8"
}
```

Possible OAuth2.0 login flows

OpenId and openId Connect are slightly different. OpenID Connect is a newer version and easier to use. OpenID is a layer on top of oauth[4] These flows are OAuth2.0 flows.

---

[2] "Identity, Claims, & Tokens – An OpenID Connect Primer, Part 1 of 3 ...." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1. Accessed 26 Feb. 2019.

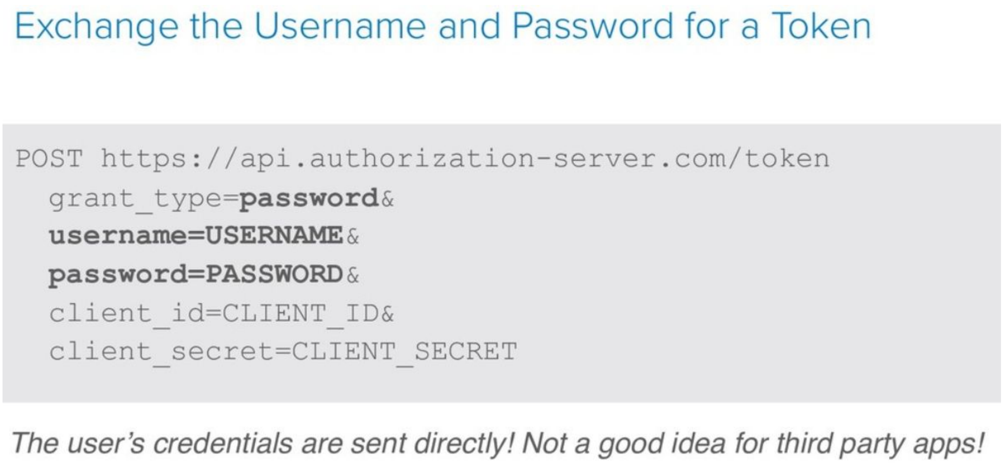
[3] "Identity, Claims, & Tokens – An OpenID Connect Primer, Part 1 of 3 ...." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1. Accessed 26 Feb. 2019.

[4] "OpenID Connect." https://openid.net/connect/. Åpnet 26 feb.. 2019.

“Flows” are openid connect and “grants” are basic oauth.[5] This is just a way of saying that oauth only handles authorization, not authentication, which we use oauth for.
Front channel: Data sent via browser
Back channel: Data sent directly to/from app

Password Grant (OAuth 2.0 Client Credentials Grant)

App has to ask the user for the user’s password. Also gives you an access token as an end result.



Exchange the Username and Password for a Token

```
POST https://api.authorization-server.com/token
  grant_type=password&
  username=USERNAME&
  password=PASSWORD&
  client_id=CLIENT_ID&
  client_secret=CLIENT_SECRET
```

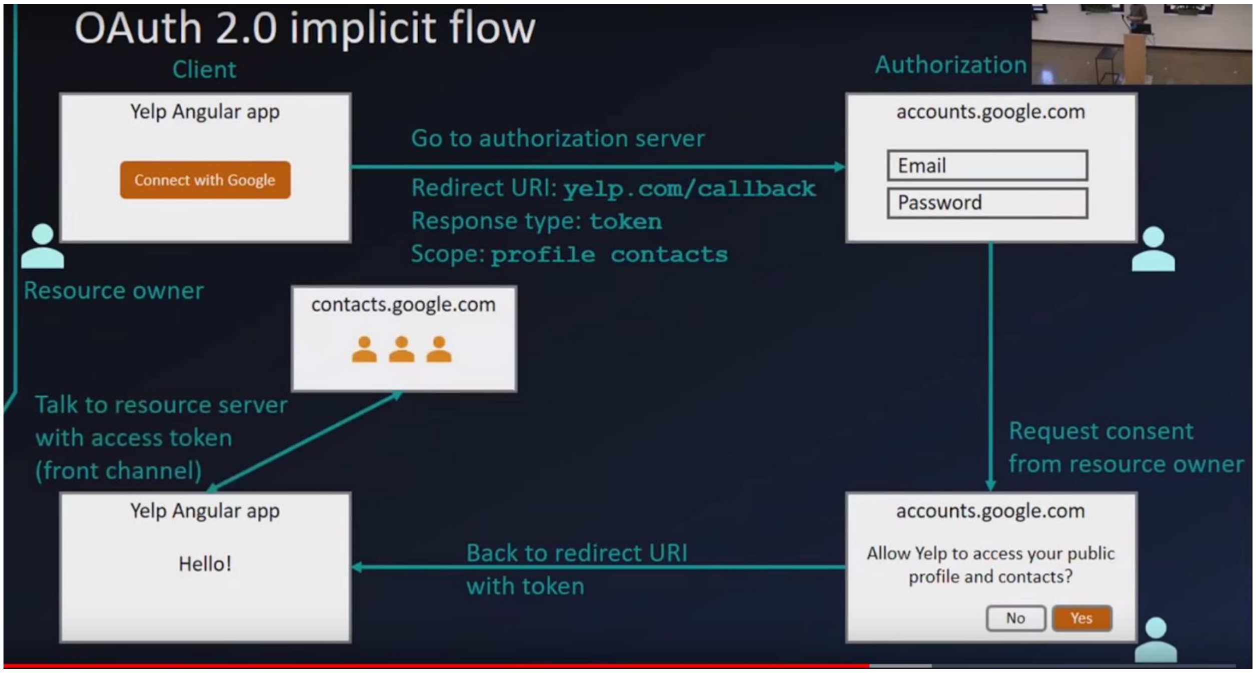
The user’s credentials are sent directly! Not a good idea for third party apps! [6]

---

[5] "When To Use Which (OAuth2) Grants and (OIDC) Flows – Robert ...." 21 mai. 2017, https://medium.com/@robert.broeckelmann/when-to-use-which-oauth2-grants-and-oidc-flows-ec6a5c00d864. Åpnet 26 feb.. 2019.

[6] "CNPDX June: OAuth all the things! | Meetup." 20 Jun. 2018, https://www.meetup.com/Cloud-Native-PDX/events/251445604/. Accessed 20 Feb. 2019.

Implicit Flow (OAuth 2.0 Implicit Grant)



OAuth 2.0 implicit flow

Client

Authorization

Yelp Angular app

Connect with Google

Go to authorization server

Redirect URI: `yelp.com/callback`
Response type: `token`
Scope: `profile contacts`

accounts.google.com

Email

Password

Resource owner

contacts.google.com

Talk to resource server
with access token
(front channel)

Request consent
from resource owner

Yelp Angular app

Hello!

Back to redirect URI
with token

accounts.google.com

Allow Yelp to access your public profile and contacts?

No Yes

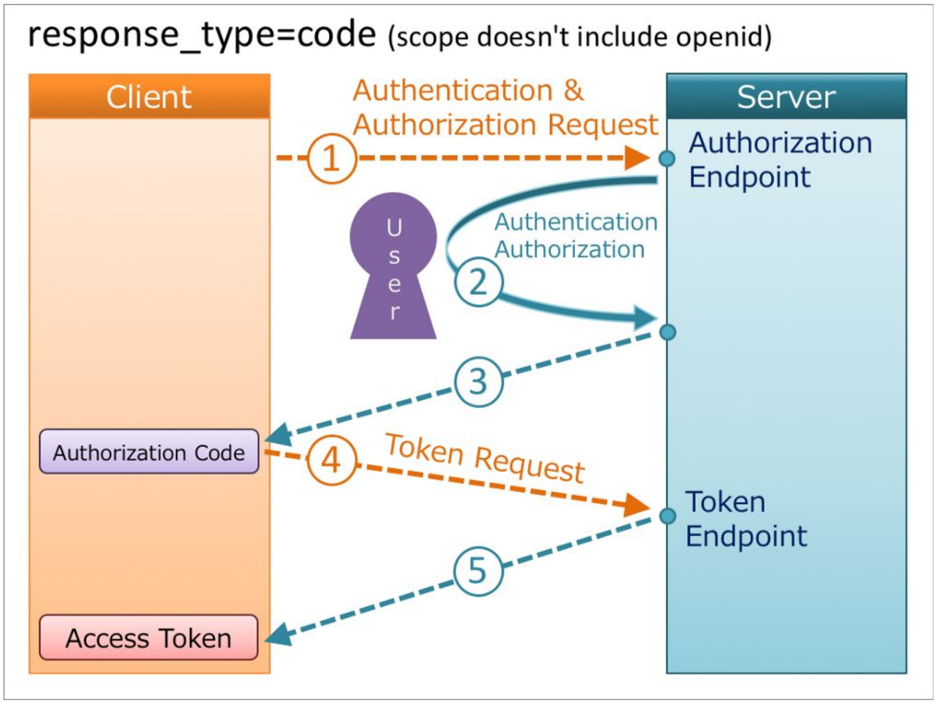
When to use Implicit Flow?

“More okay to use with OpenID Connect, not with OAuth2”. Source:
https://www.youtube.com/watch?v=wA4kqKFua2Q
https://oauth.net/2/grant-types/implicit/

https://brockallen.com/2019/01/03/the-state-of-the-implicit-flow-in-oauth2/

OAuth2.0 Authorization Code Grant

response_type=code (with scope != openid)


response_type=code (scope doesn't include openid)
Client
Server
Authentication & Authorization Request
①
Authorization Endpoint
User
Authentication Authorization
②
③
Authorization Code
④
Token Request
Token Endpoint
⑤
Access Token

“Authorization code flow on the other hand only returns a so called authorization code via the unauthenticated front channel, and requires client authentication using client id and secret (or some other mechanism) to retrieve the actual tokens (including a refresh token) via the back channel. This mechanism was originally designed for server-based applications only, since storing client secrets on a client device is questionable without the right security mechanisms in place.” Source:
https://leastprivilege.com/2014/10/10/openid-connect-hybrid-flow-and-identityserver-v3/
Leastprivilege is the creator of Identityserver3, which we are using
https://github.com/identityserver

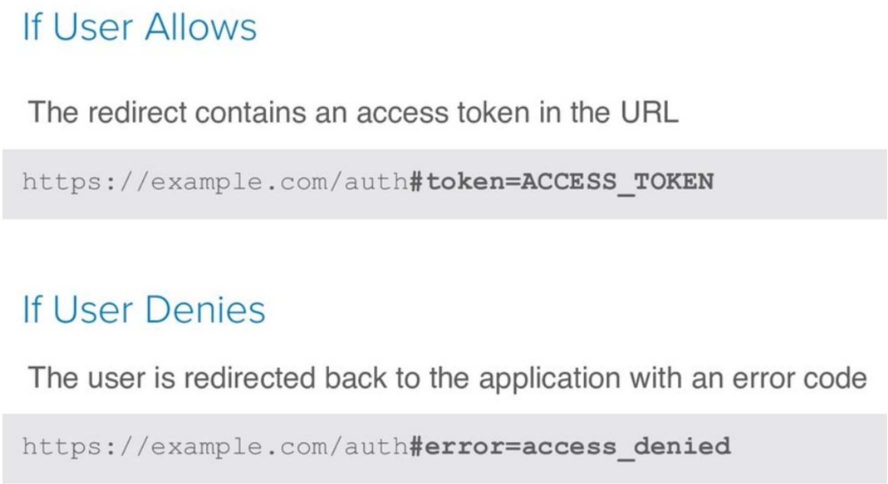
Redirect URI

While redirect URIs using localhost (i.e., "http://localhost:{port}/{path}") function similarly to loopback IP redirects described in Section 7.3, the use of localhost is NOT RECOMMENDED. Specifying a redirect URI with the loopback IP literal rather than localhost avoids inadvertently listening on network interfaces other than the loopback interface. It is also less susceptible to client-side firewalls and misconfigured host name resolution on the user's device.[7]

If an attacker can manipulate the redirect URL before the user reaches the authorization server, they could cause the server to redirect the user to a malicious server which would send the authorization code to the attacker. For public clients without a client_secret, all that is needed is the client_id and authorization code to obtain an access token. If an attacker can obtain an authorization code, they could then exchange it for an access token for public clients.[8]
This is avoided by having the redirect_uri stored in the server beforehand.

Acces token returned are in Fragment in stead of Query String (after #)

- Why?



If User Allows

The redirect contains an access token in the URL

```
https://example.com/auth#token=ACCESS_TOKEN
```

If User Denies

The user is redirected back to the application with an error code

```
https://example.com/auth#error=access_denied
```

Source: https://www.youtube.com/watch?v=wA4kqKFua2Q

Other requirements for the redirect URI/URL:

- Registered redirect URLs may contain query string parameters, but must not contain anything in the fragment.[9]
- Note that for native and mobile apps, the platform may allow a developer to register a URL scheme such as myapp:// which can then be used in the redirect URL. This means the authorization server should allow arbitrary URL schemes to be registered in order to support registering redirect URLs for native apps.
- The server should reject any authorization requests with redirect URLs that are not an exact match of a registered URL.

---

[7] "RFC 8252 - OAuth 2.0 for Native Apps - IETF Tools." https://tools.ietf.org/html/rfc8252. Accessed 26 Feb. 2019.

[8] "Redirect URL Registration - OAuth 2.0 Servers." 17 Aug. 2016, https://www.oauth.com/oauth2-servers/redirect-uris/redirect-uri-registration/. Accessed 26 Feb. 2019.

[9] "Redirect URL Registration - OAuth 2.0 Servers." 17 Aug. 2016, https://www.oauth.com/oauth2-servers/redirect-uris/redirect-uri-registration/. Accessed 18 Feb. 2019.

- If a client wishes to include request-specific data in the redirect URL, it can instead use the “state” parameter to store data that will be included after the user is redirected. It can either encode the data in the state parameter itself, or use the state parameter as a session ID to store the state on the server.

For native clients:

- Depending on the platform, native apps can either claim a URL pattern, or register a custom URL scheme that will launch the application. For example, an iOS application may register a custom protocol such as myapp:// and then use a redirect_uri of myapp://callback
- **App-Claimed https URL Redirection (Android App Links[10]/iOS Universal Links)**
    - Some platforms, (Android, and iOS (9 and later)), allow the app to override specific URL patterns to launch the native application instead of a web browser. For example, an application could register https://app.example.com/auth and whenever the web browser attempts to redirect to that URL, the operating system launches the native app instead.
    - If the operating system does support claiming URLs, this method should be used. This allows the identity of the native application to be guaranteed by the operating system. If the operating system does not support this, then the app will have to use a custom URL scheme instead.[11]
    - Have to trust in whoever manages the URL registration, to make sure nobody hijacks your app’s URL.
- **Custom URL Scheme (Android deep links[12], iOS ??)**
    - Most mobile and desktop operating systems allow apps to register a custom URL scheme that will launch the app when a URL with that scheme is visited from the system browser.
    - Using this method, the native app starts the OAuth flow as normal, by launching the system browser with the standard authorization code parameters. The only difference is that the redirect URL will be a URL with the app’s custom scheme
    - When the authorization server sends the Location header intending to redirect the user to myapp://callback#token=...., the phone will launch the

---

[10] "Handling Android App Links | Android Developers." https://developer.android.com/training/app-links/. Accessed 19 Feb. 2019.

[11] "Redirect URLs for Native Apps - OAuth 2.0 Servers." 17 Aug. 2016, https://www.oauth.com/oauth2-servers/redirect-uris/redirect-uris-native-apps/. Accessed 18 Feb. 2019.

[12] "Create Deep Links to App Content | Android Developers." https://developer.android.com/training/app-links/deep-linking. Accessed 19 Feb. 2019.

    - application and the app will be able to resume the authorization process, parsing the access token from the URL and storing it internally.
- **Custom URL Scheme Namespaces**
    - Since there is no centralized method of registering URL schemes, apps have to do their best to choose URL schemes that won't conflict with each other.
    - Follow a certain pattern
    - Malicious apps could set their URL schemes to conflict on purpose.

Sikkerhetshensyn

"The OAuth 2.0 implicit grant authorization flow (defined in
Section 4.2 of OAuth 2.0 [RFC6749]) generally works with the practice
of performing the authorization request in the browser and receiving
the authorization response via URI-based inter-app communication.
However, as the implicit flow cannot be protected by PKCE [RFC7636]
(which is required in Section 8.1), the use of the Implicit Flow with
native apps is NOT RECOMMENDED."[13]

PKCE[14]

Ved andre *flows* enn implisitt vil *autoriseringsserveren* sende *klienten* en autoriseringskode som *klient* skal bruke senere for å få tak i en *accesstoken.* PKCE er laget for å hindre at en annen part får tak i autoriseringskoden fra *autoriseringsserveren*. What PKCE (not) protect against: https://web-in-security.blogspot.com/2017/01/pkce-what-cannot-be-protected.html

For every authorization request done by a client a code-verifier *must* be generated. This may just be a random string. The client then makes a secure hash of the verifier and sends the hash and hashfunction with the authorization request(this should be sent over a secure TLS connection). When the client later requests the accesstoken from the authorizationserver the code-verifier(original, plain string) is sent with the authorization code. Now the authorizationserver can run the code-verifier through the same hashfunction used by the client and compare the resulting hash with the hash from the initial request. If these don't match is likely the request for the token was sent by someone else.

If a secure hashfunction is used it will be practically impossible to make a forgery even if the attacker has access to the hash. The code-verifier *must* be kept secret.

Autoriseringsserveren

---

[13] "RFC 8252 - OAuth 2.0 for Native Apps - IETF Tools." https://tools.ietf.org/html/rfc8252. Accessed 20 Feb. 2019.

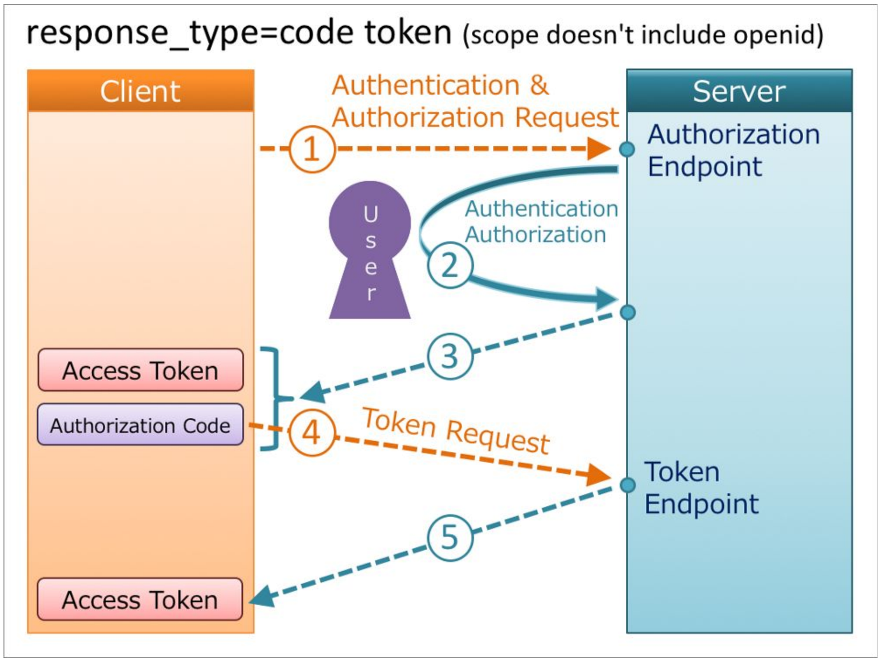
[14] "RFC 7636 - Proof Key for Code Exchange by OAuth ... - IETF Tools." https://tools.ietf.org/html/rfc7636. Accessed 20 Feb. 2019.

Kan være lurt å kreve at serveren sjekker package-names og/eller bundle ID's for appen som ber om tilgang.

Combining Implicit and Code grant?

This is possible in OAuth2.0

response_type=code token (with scope != openid)



If an Access Token is returned from both the Authorization Endpoint and from the Token Endpoint, which is the case for the response_type values code token and code id_token token, their values MAY be the same or they MAY be different. **Note that different Access Tokens might be returned be due to the different security characteristics of the two endpoints and the lifetimes and the access to resources granted by them might also be different.**[15]

But why would one use this? Stackoverflow users are baffled.[16]

We do not recommend that an Access Token obtained when response_type=code token or code token or code id_token token be used to call APIs.[17]

---

[15] "Diagrams of All The OpenID Connect Flows – Takahiko ... - Medium." 30 Oct. 2017, https://medium.com/@darutk/diagrams-of-all-the-openid-connect-flows-6968e3990660. Accessed 5 Mar. 2019.

[16] "Usage of response_type="code token" in OAuth 2? - Stack Overflow." 6 Nov. 2014, https://stackoverflow.com/questions/26744079/usage-of-response-type-code-token-in-oauth-2. Accessed 5 Mar. 2019.

[17] "How to Implement the Hybrid Flow - Auth0." https://auth0.com/docs/api-auth/tutorials/hybrid-flow. Accessed 5 Mar. 2019.

OpenID Connect

“OpenID Connect (henceforth OIDC), runs on top of OAuth 2.0.

OAuth 2.0 leaves a lot of details up to implementers. For instance, it supports scopes, but scope names are not specified. It supports access tokens, but the format of those tokens are not specified. With OIDC, a number of specific scope names are defined that each produce different results. OIDC has both access tokens and ID tokens. An ID token must be JSON web token (JWT). Since the specification dictates the token format, it makes it easier to work with tokens across implementations.

Typically, you kick off an OIDC interaction by hitting an /authorization endpoint with an HTTP GET. A number of query parameters indicate what you can expect to get back after authenticating and what you’ll have access to (authorization).

Often, you’ll need to hit a /token endpoint with an HTTP POST to get tokens which are used for further interactions.

OIDC also has an /introspect endpoint for verifying a token, a /userinfo endpoint for getting identity information about the user.”[18]

OAuth 2.0 Authentication Servers implementing OpenID Connect are also referred to as OpenID Providers (OPs). OAuth 2.0 Clients using OpenID Connect are also referred to as Relying Parties (RPs).

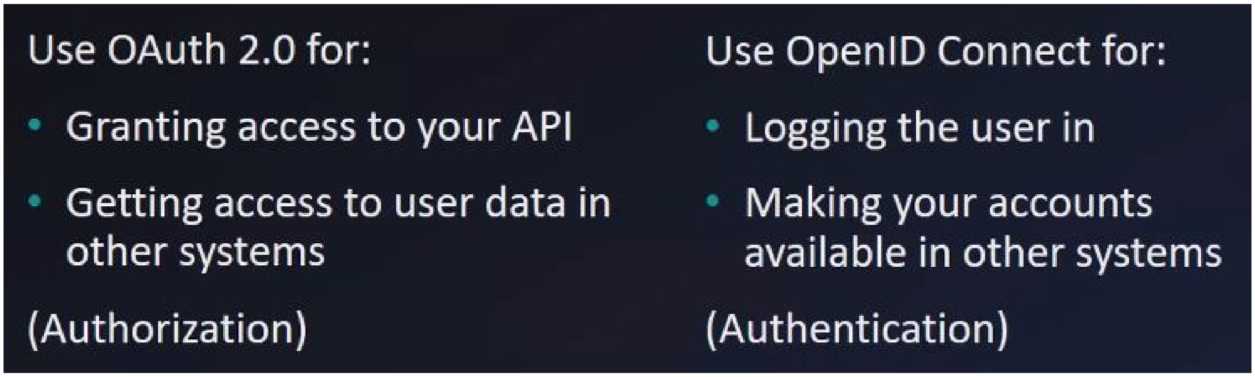
When to use OpenID Connect?

- In OAuth there is no standard way of getting the user’s information (email, name)
- OpenID Connect:
    - Adds ID token

---

[18] "Identity, Claims, & Tokens – An OpenID Connect Primer, Part 1 of 3 ...." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1. Accessed 26 Feb. 2019.

        - JSON Web Token about the user, “scrambled”, needs to be decoded to be human readable
    - Adds userinfo endpoint (If ID token does not provide enough info)
        - Use access token for this call, not ID token
    - Adds standard set of scopes and other standardization



Use OAuth 2.0 for:

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

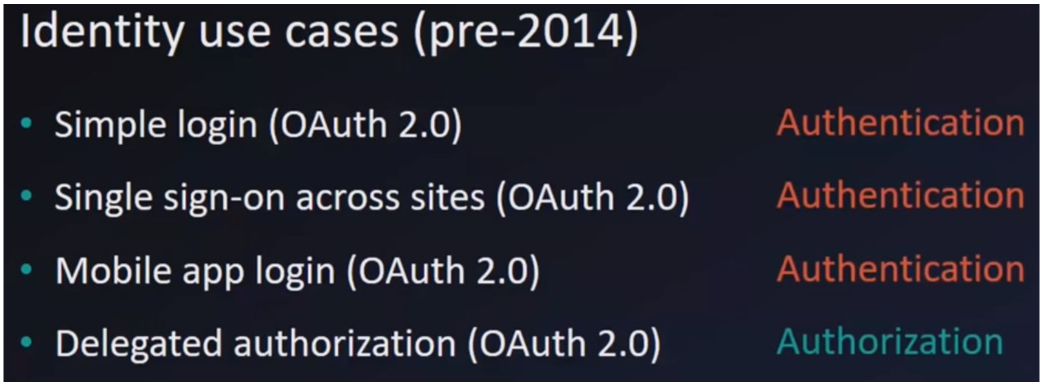
(Authorization)

Use OpenID Connect for:

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

(Authentication)

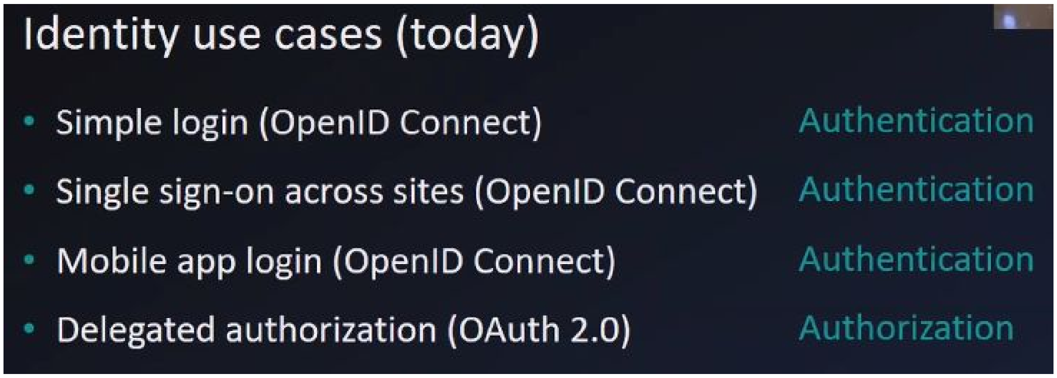
Without OpenID Connect:



Identity use cases (pre-2014)

• Simple login (OAuth 2.0)	Authentication
• Single sign-on across sites (OAuth 2.0)	Authentication
• Mobile app login (OAuth 2.0)	Authentication
• Delegated authorization (OAuth 2.0)	Authorization

With OpenID Connect:



Identity use cases (today)

- Simple login (OpenID Connect) Authentication
- Single sign-on across sites (OpenID Connect) Authentication
- Mobile app login (OpenID Connect) Authentication
- Delegated authorization (OAuth 2.0) Authorization

More on OAuth2 and where we got the pictures from:
https://www.youtube.com/watch?v=996OiexHze0

OpenID Connect Tokens

There are three types of tokens in OIDC: `id_token`, `access_token` and `refresh_token`.

ID Tokens

An `id_token` is a JWT (JSON Web Token), per the OIDC Specification. This means that:

- identity information about the user is encoded right into the token and
- the token can be definitively verified to prove that it hasn't been tampered with.

There's a set of rules in the specification for validating an `id_token`. Among the claims encoded in the `id_token` is an expiration (`exp`), which must be honored as part of the validation process. Additionally, the signature section of JWT is used in concert with a key to validate that the entire JWT has not been tampered with in any way.[19]

---

[19] "Identity, Claims, & Tokens – An OpenID Connect Primer, Part 1 of 3 ...." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1. Accessed 26 Feb. 2019.

JWT

In 2015, the JWT spec was released. The includes provisions for cryptographically signed JWTs (called JWSs). A signed JWT is particularly useful in application development because you can have a high degree of confidence that the information encoded into the JWT has not been tampered with. By verifying the JWT within the application, you can avoid another round trip to an API service. OIDC formalizes the role of JWT in mandating that ID Tokens be JWTs.

ID Token verification

ID tokens are sensitive and can be misused if intercepted. You must ensure that these tokens are handled securely by transmitting them only over HTTPS and only via POST data or within request headers. If you store them on your server, you must also store them securely.

One thing that makes ID tokens useful is that fact that you can pass them around different components of your app. These components can use an ID token as a lightweight authentication mechanism authenticating the app and the user. But before you can use the information in the ID token or rely on it as an assertion that the user has authenticated, you **must** validate it.[20]

OpenID Connect Flows

"response_type" value	**Flow**
`code`	Authorization Code Flow
`id_token`	Implicit Flow
`id_token token`	Implicit Flow
`code id_token`	Hybrid Flow
`code token`	Hybrid Flow

---

[20] "OpenID Connect | Google Identity Platform | Google Developers." 20 Dec. 2018, https://developers.google.com/identity/protocols/OpenIDConnect. Accessed 27 Feb. 2019.

`code id_token token`	Hybrid Flow[21]

Authorization Code, *Implicit*, and *Hybrid*. These flows are controlled by the `response_type` query parameter in the `/authorization` request. When thinking of which flow to use, consider front-channel vs. back-channel requirements. Front-channel refers to a user-agent (such as a Single-Page-Application or mobile app) interacting directly with the OpenID provider (OP). The implicit flow is a good choice when front-channel communication is required. Back-channel refers to a middle-tier client (below presentation layer, above data layer, such as Spring Boot or Express) interacting with the OP. The authorization code flow is a good choice when back-channel communication is required.[22]

OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. Use of this extension is requested by Clients by including the `openid` scope value in the Authorization Request. Information about the authentication performed is returned in a **JSON Web Token (JWT)** [JWT] called an ID Token (see **Section 2**). [23]

---

[21] "Final: OpenID Connect Core 1.0 incorporating errata set 1." 8 Nov. 2014, https://openid.net/specs/openid-connect-core-1_0.html. Accessed 5 Mar. 2019.

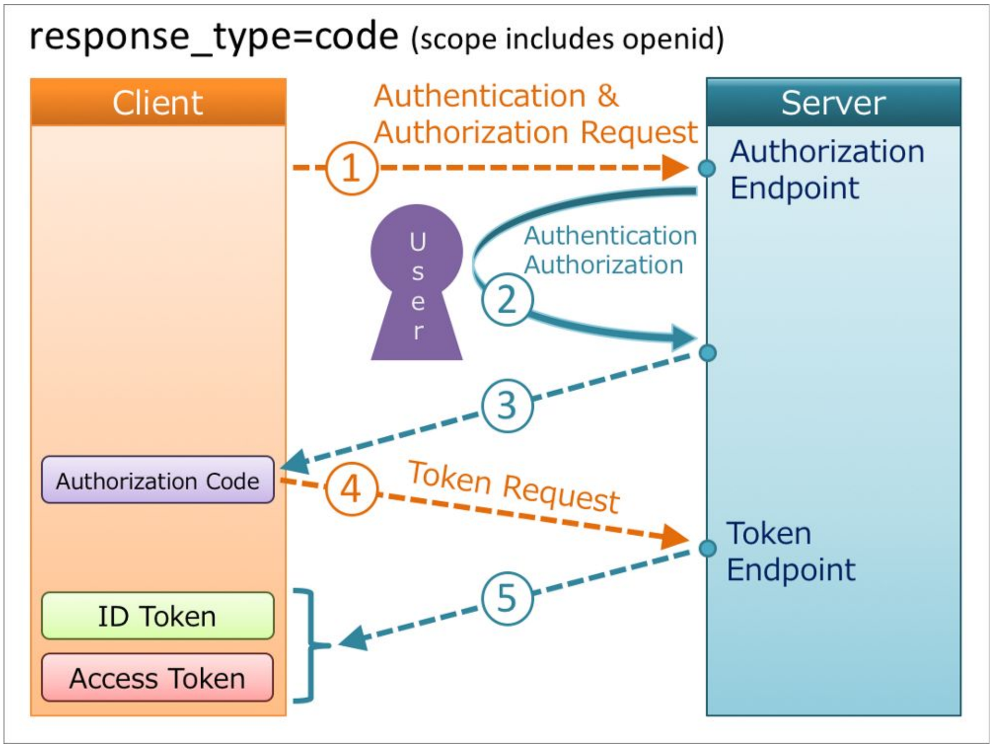
[22] "Identity, Claims, & Tokens – An OpenID Connect Primer, Part 1 of 3 ...." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1. Accessed 26 Feb. 2019.

[23] "Final: OpenID Connect Core 1.0 incorporating errata set 1." 8 Nov. 2014, https://openid.net/specs/openid-connect-core-1_0.html. Accessed 5 Mar. 2019.

OpenID Connect Authorization Code Flow

This is a suitable approach when you have a middleware client connected to an OIDC OP and don't (necessarily) want tokens to ever come back to an end-user application, such as a browser. It also means the end-user application never needs to know a secret key.[24]

response_type=code (with scope=openid):


response_type=code (scope includes openid)
Client
Server
Authentication & Authorization Request
1
Authorization Endpoint
User
Authentication Authorization
2
3
Authorization Code
4
Token Request
Token Endpoint
5
ID Token
Access Token

25

Authorization Code flow uses `response_type=code`. After successful authentication, the response will contain a `code` value. This code can later be exchanged for an `access_token` and an `id_token` (Hang in for now, we'll talk about tokens in more depth later on.) This flow is useful where you have "middleware" as part of the architecture. The middleware has a `client_id` and `client_secret`, which is required to exchange the `code` for tokens by hitting the `/token` endpoint. These tokens can then be returned to the end-user application, such as a browser, without the browser ever having to know the `client_secret`. This flow allows for long-lived

---

[24] "OIDC in Action – An OpenID Connect Primer, Part 2 ... - Okta Developer." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-2. Accessed 26 Feb. 2019.

[25] "Diagrams of All The OpenID Connect Flows – Takahiko ... - Medium." 30 Oct. 2017, https://medium.com/@darutk/diagrams-of-all-the-openid-connect-flows-6968e3990660. Accessed 26 Feb. 2019.

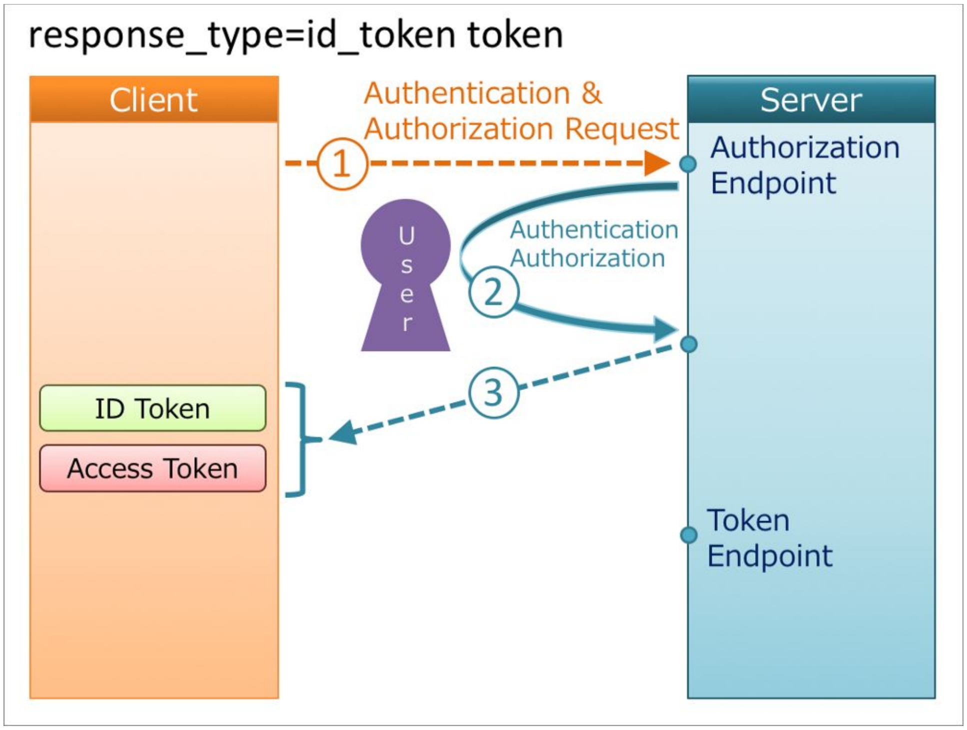
sessions through the use of `refresh tokens`. The only purpose of `refresh tokens` is to obtain new `access tokens` to extend a user session.[26]

That code can now be exchanged for an `id_token` and an `access_token` by the middle tier - a Spring Boot application, in this case. This middle tier will validate the state we sent in the authorize request earlier and make a `/token` request using the Client Secret to mint an `access_token` and `id_token` for the user.[27]

OpenID Connect Implicit Flow

This is a suitable approach when working with a client (such as a Single Page Application or mobile application) that you want to interact with the OIDC OP directly.[28]

response_type=id_token token



Implicit flow uses `response_type=id_token token` or `response_type=id_token`. After successful authentication, the response will contain

---

[26] "Identity, Claims, & Tokens – An OpenID Connect Primer, Part 1 of 3 ...." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1. Accessed 26 Feb. 2019.
[27] "OIDC in Action – An OpenID Connect Primer, Part 2 ... - Okta Developer." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-2. Accessed 26 Feb. 2019.
[28] "OIDC in Action – An OpenID Connect Primer, Part 2 ... - Okta Developer." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-2. Accessed 26 Feb. 2019.

an `id_token` and an `access_token` in the first case or just an `id_token` in the second case. This flow is useful when you have an app speaking directly to a backend to obtain tokens with no middleware. It does not support long-lived sessions.[29]

The application can now verify the `id_token` locally. Use the `/introspect` endpoint to verify the `access_token`. It can also use the `access_token` as a bearer token to hit protected resources, such as the `/userinfo` endpoint.[30]

OpenID Connect Hybrid Flow

The Hybrid Flow is an OpenID Connect (OIDC) grant that enables use cases where your application can immediately use an ID token to access information about the user while obtaining an authorization code that can be exchanged for an Access Token (therefore gaining access to protected resources for an extended period of time).[31]

Hybrid flow (as the name indicates) is a combination of the above two. It allows to request a combination of identity token, access token and code via the front channel using either a fragment encoded redirect (native and JS based clients) or a form post (server-based web applications). This enables e.g. scenarios where your client app can make immediate use of an identity token to get access to the user's identity but also retrieve an authorization code that that can be used (e.g. by a back end service) to request a refresh token and thus gaining long lived access to resources.[32]

This is a suitable approach when you want your end-user application to have immediate access to short-lived tokens – such as the `id_token` for identity information, and also want to use a backend service to exchange the authorization code for longer-lived tokens using refresh tokens.[33]

Leastprivilege is the creator of Identityserver3, which we are using https://github.com/identityserver

---

[29] "Identity, Claims, & Tokens – An OpenID Connect Primer, Part 1 of 3 ...." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1. Accessed 26 Feb. 2019.

[30] "OIDC in Action – An OpenID Connect Primer, Part 2 ... - Okta Developer." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-2. Accessed 26 Feb. 2019.

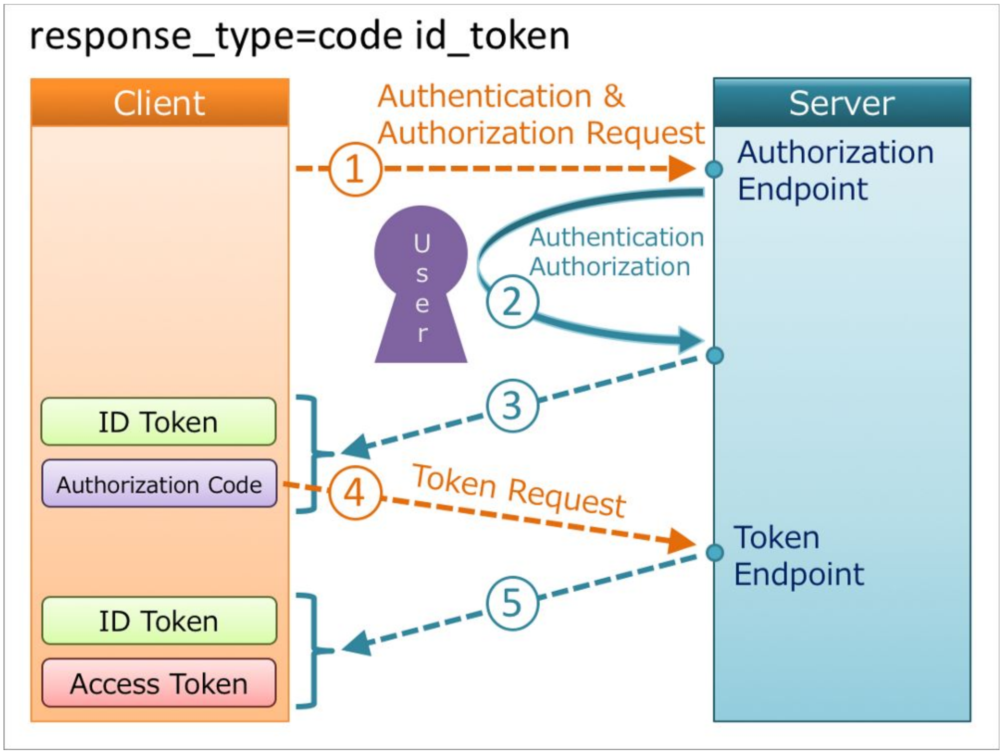
[31] "How to Implement the Hybrid Flow - Auth0." https://auth0.com/docs/api-auth/tutorials/hybrid-flow. Accessed 27 Feb. 2019.

[32] "OpenID Connect Hybrid Flow and IdentityServer v3 | leastprivilege.com." 10 okt.. 2014, https://leastprivilege.com/2014/10/10/openid-connect-hybrid-flow-and-identityserver-v3/. Åpnet 26 feb.. 2019.

[33] "OIDC in Action – An OpenID Connect Primer, Part 2 ... - Okta Developer." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-2. Accessed 26 Feb. 2019.

When an access token is issued together with an ID token from the authorization endpoint, the hash value of the access token calculated in a certain way has to be embedded in the ID token. So, be careful when you implement this flow.[34] See "3.2.2.10 ID Token" in OpenID Connect Core 1.0

response_type=code id_token (OUR Hybrid Flow)



When receiving ID Tokens from both endpoints, their overlapping fields must match.

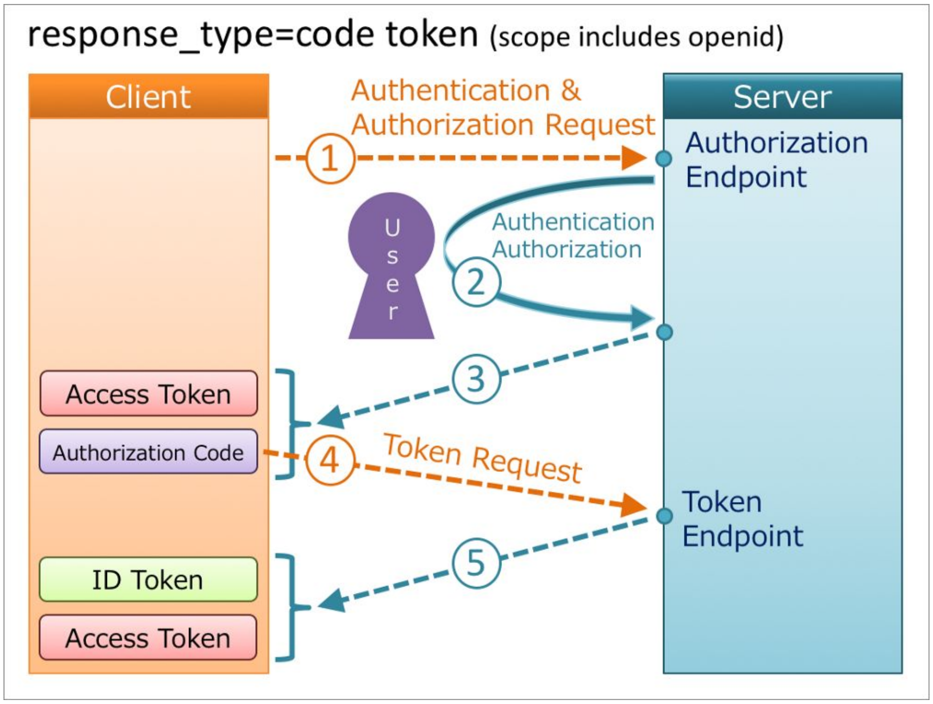
When an authorization code is issued together with an ID token from the authorization endpoint, the hash value of the authorization code calculated in a certain way has to be embedded in the ID token.[35]

---

[34] "Diagrams of All The OpenID Connect Flows – Takahiko ... - Medium." 30 Oct. 2017, https://medium.com/@darutk/diagrams-of-all-the-openid-connect-flows-6968e3990660. Accessed 26 Feb. 2019.

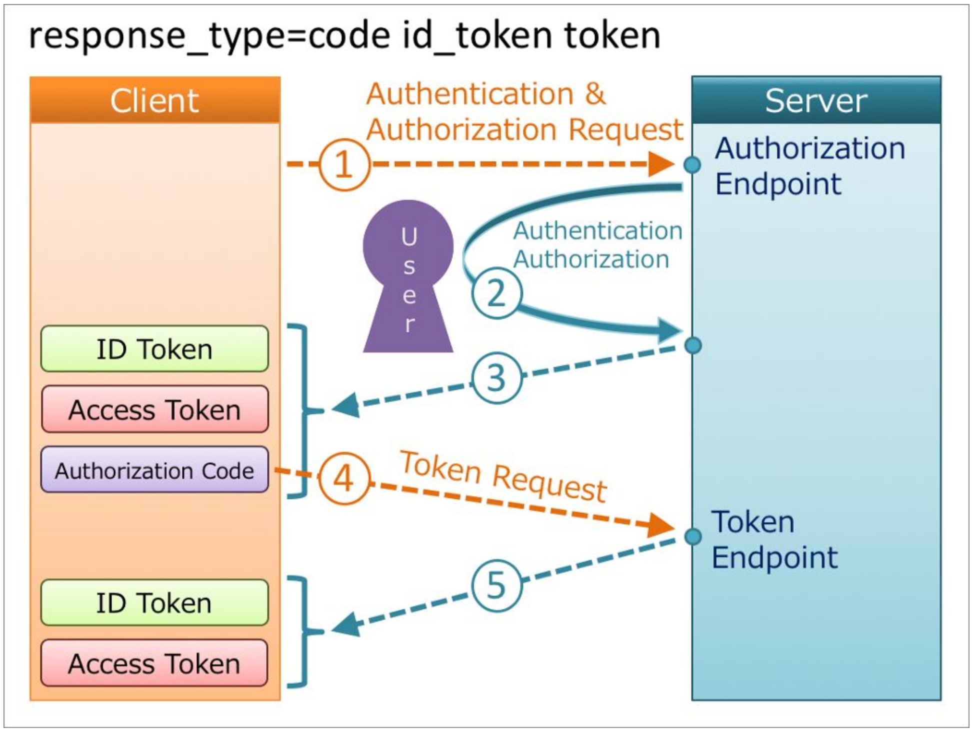
[35] "Diagrams of All The OpenID Connect Flows – Takahiko ... - Medium." 30 Oct. 2017, https://medium.com/@darutk/diagrams-of-all-the-openid-connect-flows-6968e3990660. Accessed 26 Feb. 2019.

response_type=code token (with scope = openid) (Hybrid Flow)



Vi skal ha ID token, så vi bruker denne.

response_type=code id_token token (Hybrid flow)



The Access Code step

When using the Hybrid Flow, Token Requests are made in the same manner as for the Authorization Code Flow, as defined in **Section 3.1.3.1** of the OpenID Connect spec.

Section 3.1.3.1:
A Client makes a Token Request by presenting its Authorization Grant (in the form of an Authorization Code) to the Token Endpoint using the `grant_type` value `authorization_code`, as described in Section 4.1.3 of **OAuth 2.0** [RFC6749].

Section 4.1.3 of **OAuth 2.0**

The client makes a request to the token endpoint by sending the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
REQUIRED. Value MUST be set to "authorization_code".

code
REQUIRED. The authorization code received from the authorization server.

redirect_uri
REQUIRED, if the "redirect_uri" parameter was included in the authorization request as described in Section 4.1.1, and their values MUST be identical.

client_id
REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using TLS (with extra line breaks for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

OAuth2.0 **3.2.1**. Client Authentication

Confidential clients or other clients issued client credentials MUST authenticate with the authorization server as described in Section 2.3 when making requests to the token endpoint.

A client MAY use the "client_id" request parameter to identify itself when sending requests to the token endpoint. In the "authorization_code" "grant_type" request to the token endpoint, an unauthenticated client MUST send its "client_id" to prevent itself from inadvertently accepting a code intended for a client with a different "client_id". This protects the client from substitution of the authentication code. (It provides no additional security for the protected resource.)

OAuth2.0 **2.3**. Client Authentication

If the client type is confidential, the client and authorization server establish a client authentication method suitable for the security requirements of the authorization server. The authorization server MAY accept any form of client authentication meeting its security requirements.

Confidential clients are typically issued (or establish) a set of client credentials used for authenticating with the authorization server (e.g., password, public/private key pair).

The authorization server MAY establish a client authentication method with public clients. However, the authorization server MUST NOT rely on public client authentication for the purpose of identifying the client.

And this is where OAuth2.0 stops. What about this?:

Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

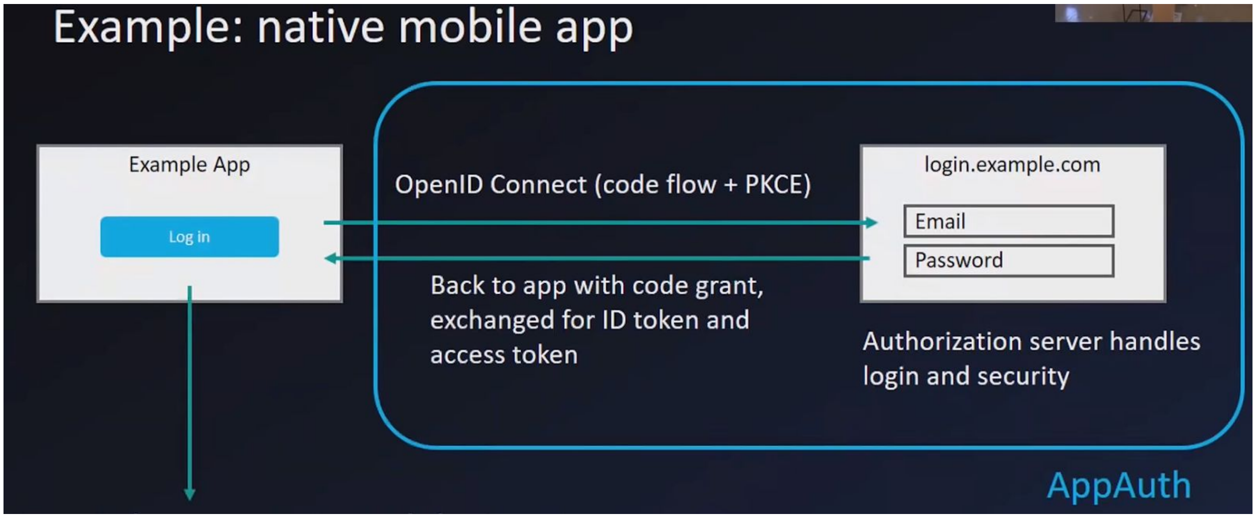
Lucky for us, we found this guidance from Oracle:

“This request must authenticate using HTTP basic. Use your app’s Client Id as the username and its Client Secret as the password. The format is *client_id:client_secret*. Encode the string with base-64 encoding, and you can pass it as an authentication header. The system does not support passing Client Id and Client Secret parameters in the JSON body, and, unlike basic authentication elsewhere, you should not include your site name."[36]

Other OpenID Connect flows

There are two other flows not covered here: Client Credentials Flow and Resource Owner Password Credentials. These are both defined in the OAuth 2.0 spec and, as such, are supported by OIDC. Here, we're focusing on flows that require an external authentication provider, such as Okta or Google, and not the alternative methods that these flows support.[37]

Recommended flow for native mobile app:



Example: native mobile app

OpenID Connect (code flow + PKCE)

Back to app with code grant, exchanged for ID token and access token

Authorization server handles login and security

Authorization code flow with PKCE

- Recommended: Use AppAuth, BUT, we can't have dependencies so, no.
- Proof Code For Key Exchange (PKCE)
    - "On the fly client secret" (Source: OAuth all the Things! What is OAuth 2.0?)
- Store tokens in protected device storage
- Use ID token to know who the user is
- Attach access token to outgoing API requests

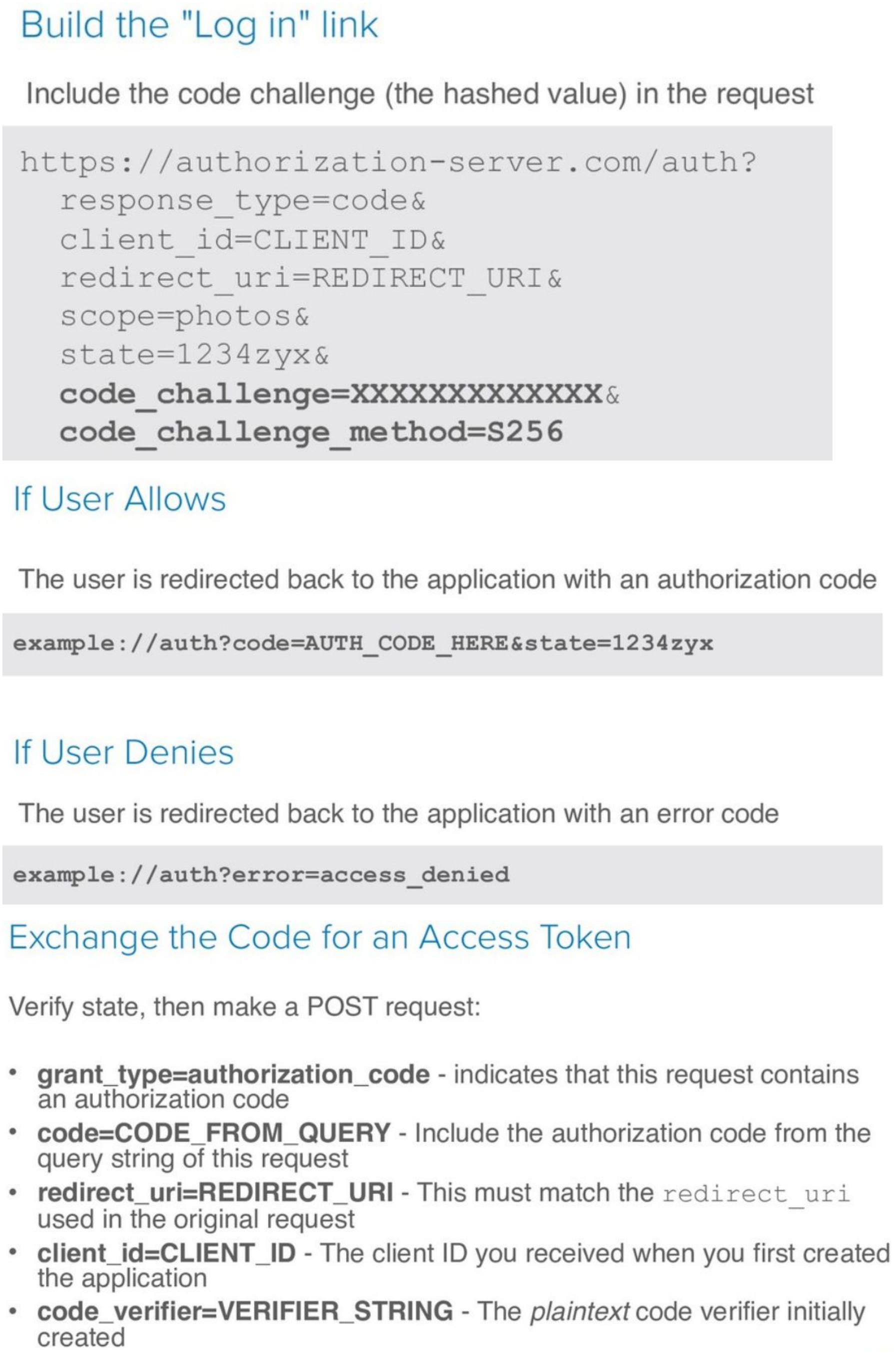
Source: https://www.youtube.com/watch?v=996OiexHze0

---

[36] "Authenticate Using OAuth 2.0 - Oracle Docs." https://docs.oracle.com/cloud/latest/marketingcs_gs/OMCAB/Developers/GettingStarted/Authentication/authenticate-using-oauth.htm. Accessed 5 Mar. 2019.

[37] "OIDC in Action – An OpenID Connect Primer, Part 2 ... - Okta Developer." 25 Jul. 2017, https://developer.okta.com/blog/2017/07/25/oidc-primer-part-2. Accessed 26 Feb. 2019.

How this works in practice:



Build the "Log in" link

Include the code challenge (the hashed value) in the request

```
https://authorization-server.com/auth?
  response_type=code&
  client_id=CLIENT_ID&
  redirect_uri=REDIRECT_URI&
  scope=photos&
  state=1234zyx&
  code_challenge=XXXXXXXXXXXXX&
  code_challenge_method=S256
```

If User Allows

The user is redirected back to the application with an authorization code

```
example://auth?code=AUTH_CODE_HERE&state=1234zyx
```

If User Denies

The user is redirected back to the application with an error code

```
example://auth?error=access_denied
```

Exchange the Code for an Access Token

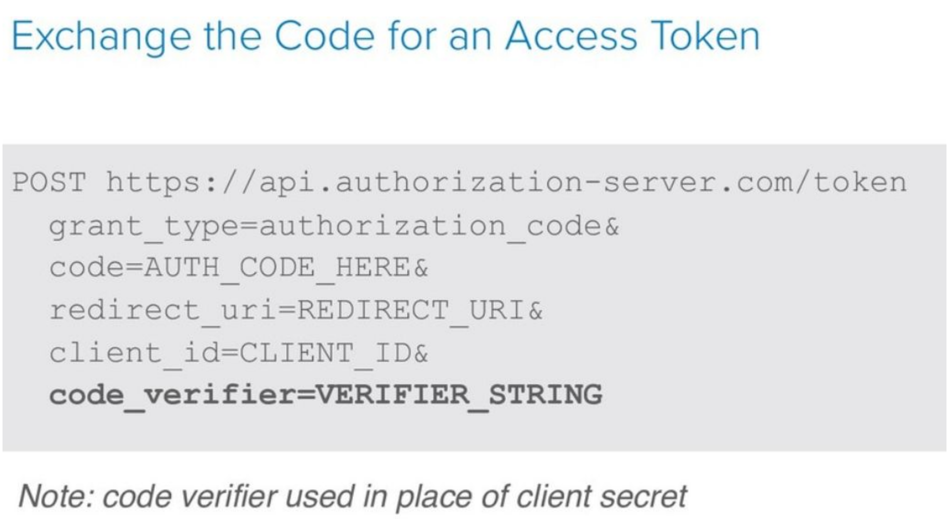
Verify state, then make a POST request:

- **grant_type=authorization_code** - indicates that this request contains an authorization code
- **code=CODE_FROM_QUERY** - Include the authorization code from the query string of this request
- **redirect_uri=REDIRECT_URI** - This must match the `redirect_uri` used in the original request
- **client_id=CLIENT_ID** - The client ID you received when you first created the application
- **code_verifier=VERIFIER_STRING** - The *plaintext* code verifier initially created

When the native application then exchanges the code for the access token (Step 8 above), it will include the code_verifier string on that call.[38]

---

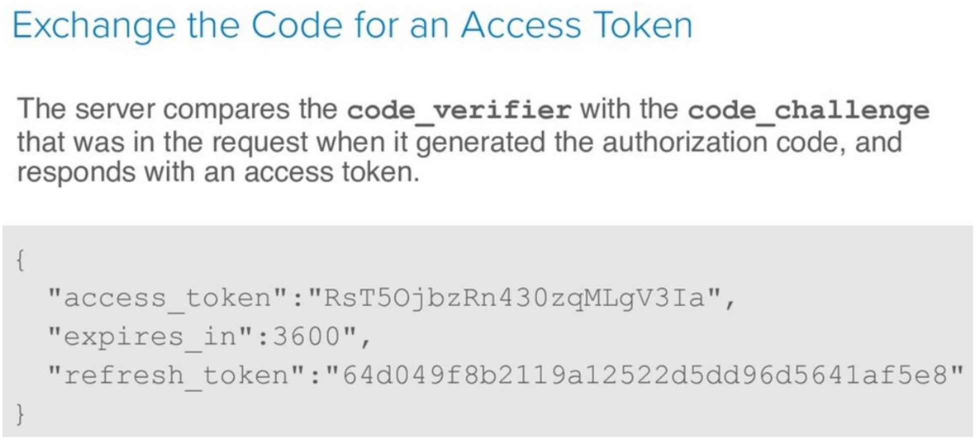
[38] "Enhancing OAuth Security for Mobile Applications with PKCE – OpenID." 26 May. 2015, https://openid.net/2015/05/26/enhancing-oauth-security-for-mobile-applications-with-pkse/. Accessed 22 Feb. 2019.



Exchange the Code for an Access Token

```
POST https://api.authorization-server.com/token
  grant_type=authorization_code&
  code=AUTH_CODE_HERE&
  redirect_uri=REDIRECT_URI&
  client_id=CLIENT_ID&
  code_verifier=VERIFIER_STRING
```

Note: code verifier used in place of client secret



Exchange the Code for an Access Token

The server compares the `code_verifier` with the `code_challenge` that was in the request when it generated the authorization code, and responds with an access token.

```
{
  "access_token":"RsT5OjbzRn430zqMLgV3Ia",
  "expires_in":3600",
  "refresh_token":"64d049f8b2119a12522d5dd96d5641af5e8"
}
```

Doing this without PKCE:
http://www.thread-safe.com/2014/05/the-correct-use-of-state-parameter-in.html
Mentioned here (newest unpublished OAuth2 security RFC):
https://tools.ietf.org/html/draft-ietf-oauth-security-topics-11#section-2.1.1
Possibly:
https://rograce.github.io/openid-connect-documentation/explore_auth_code_flow
https://developer.okta.com/authentication-guide/implementing-authentication/auth-code-pkce

https://docs.microsoft.com/en-us/azure/active-directory/develop/v1-protocols-openid-connect-code

“Oauth 2.0 for native apps”, RFC
https://tools.ietf.org/html/rfc8252

Diskusjon rundt SSO
Del 1, OAuth2

https://medium.com/@adrianmihaila/single-sign-on-across-multiple-applications-part-i-93fb0616ddc8
Del 2:
OAuth2 + OpenID Connect (Gir info om den som sender requests)
https://medium.com/@adrianmihaila/single-sign-on-across-multiple-applications-part-ii-a8a48e4a7c11

OpenID Connect Core Spec: https://openid.net/specs/openid-connect-core-1_0.htm
How Google does OpenID: https://developers.google.com/identity/protocols/OpenIDConnect
OAuth Playground: https://www.oauth.com/playground/index.html