

Get intimate with OAuth

Milan Starčević | 02.11.2023 | Heapcon

Agenda

- 
- 01** | The Why? (The problem OAuth solves)
 - 02** | Learn the names (Terminology)
 - 03** | Going with the flow (How to choose an OIDC Flow)
 - 04** | Mind the gap (Implementing and common pitfalls)

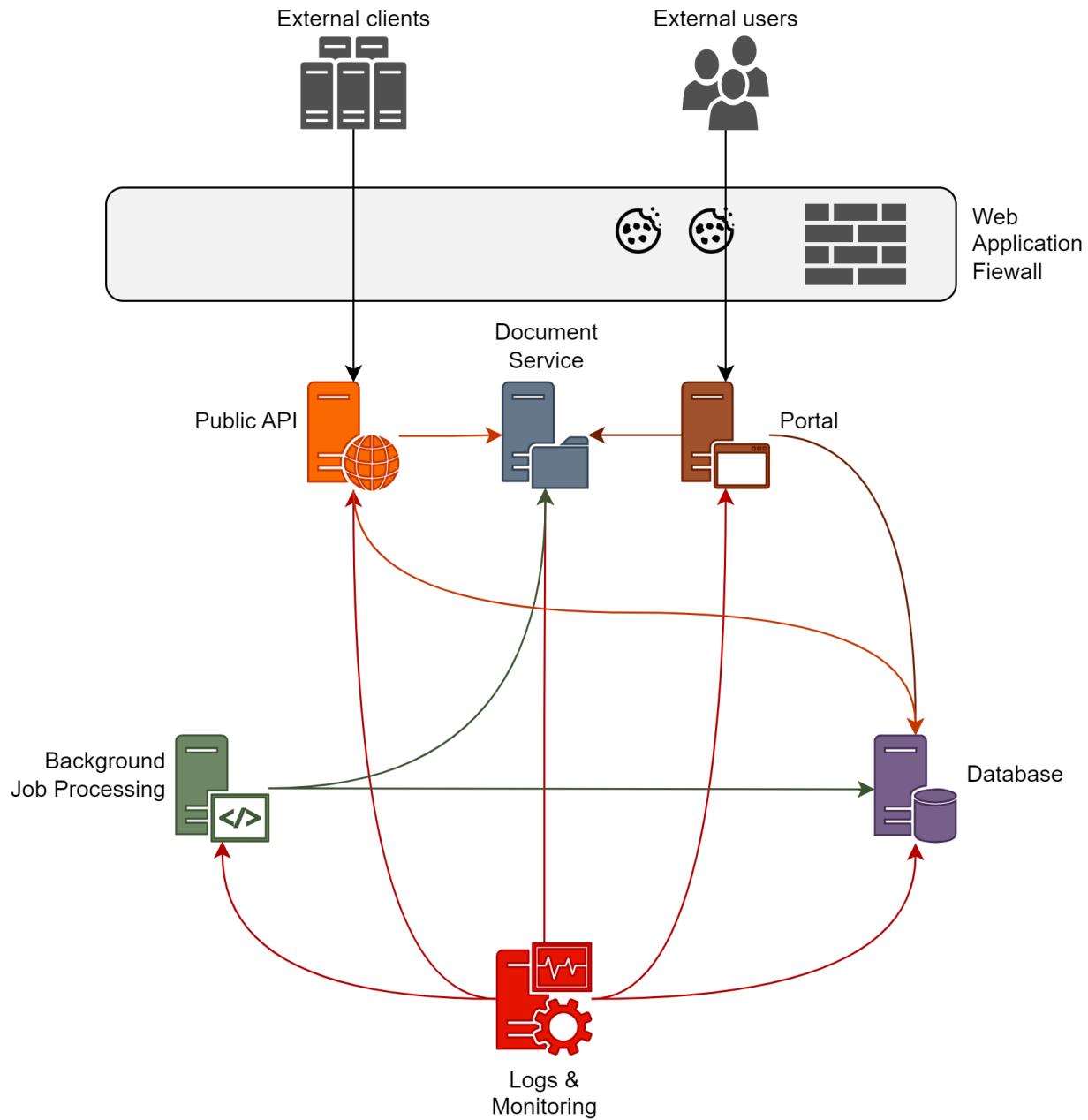
01



The Why

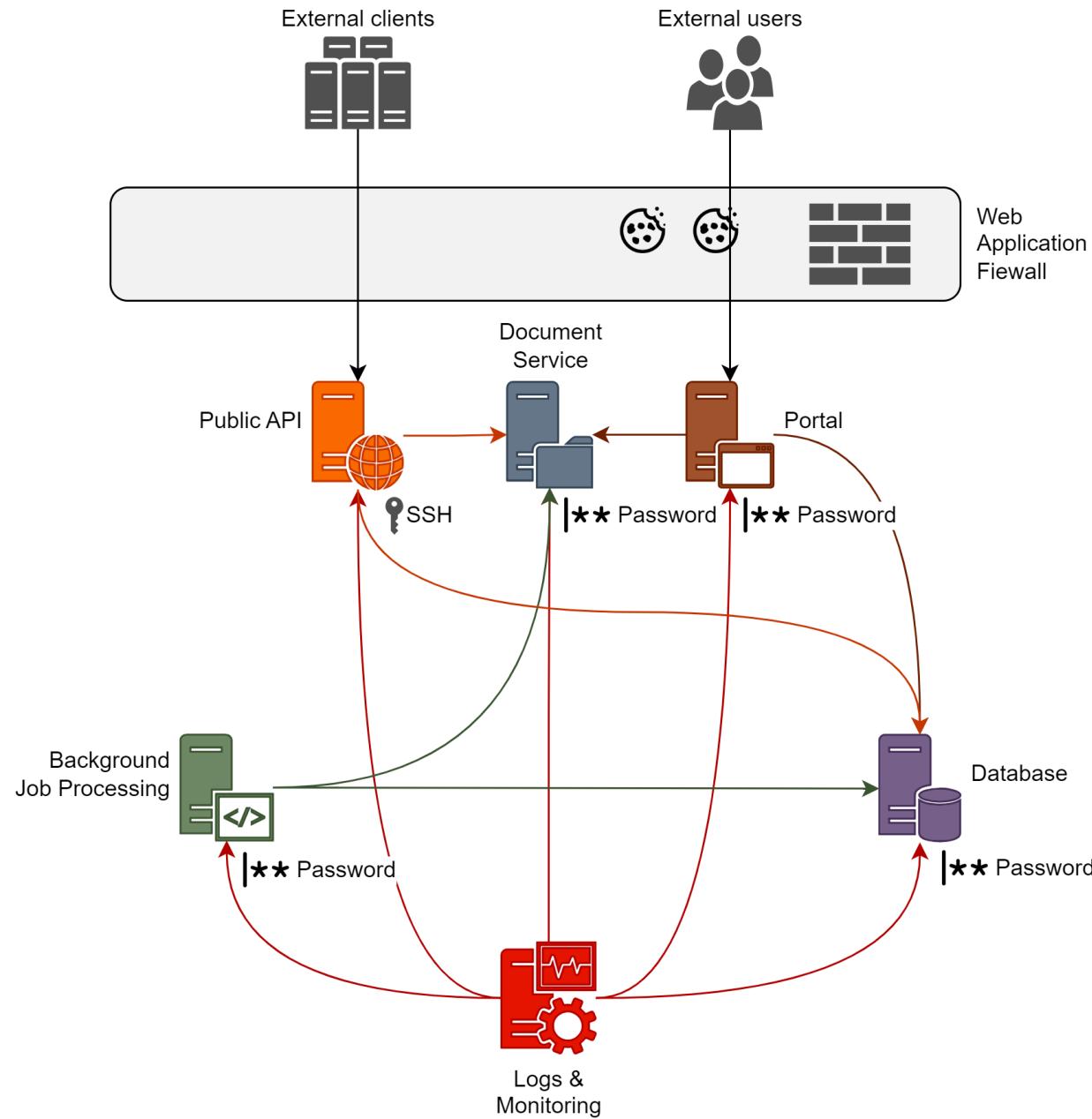
The problem OAuth solves

System



Authentication

1. Each component must authenticate its clients in order to trust them
2. Different authentication mechanism





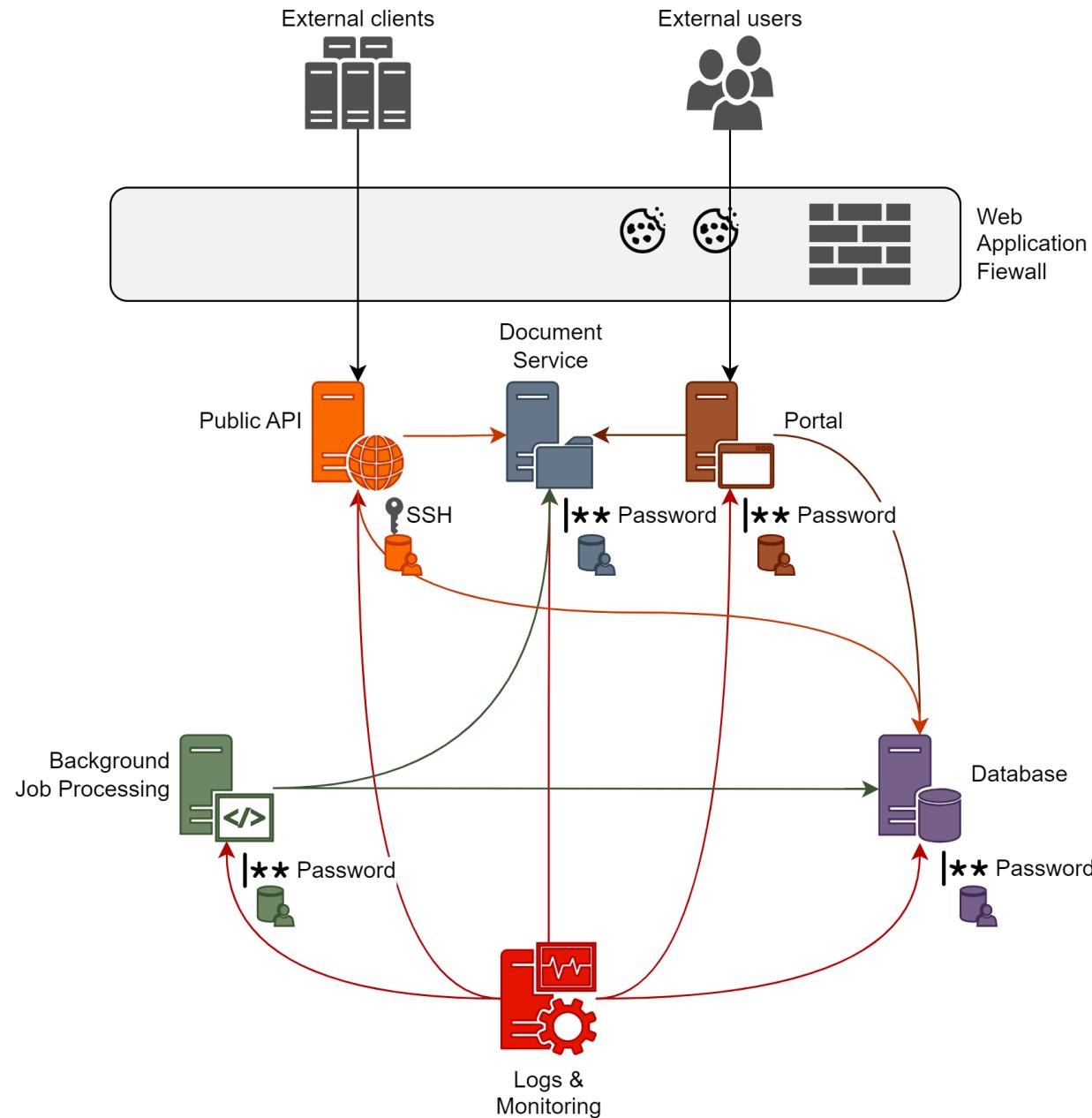
imgflip.com

No single source of truth

That means:

Each component has
to keep a separate
identity stores.

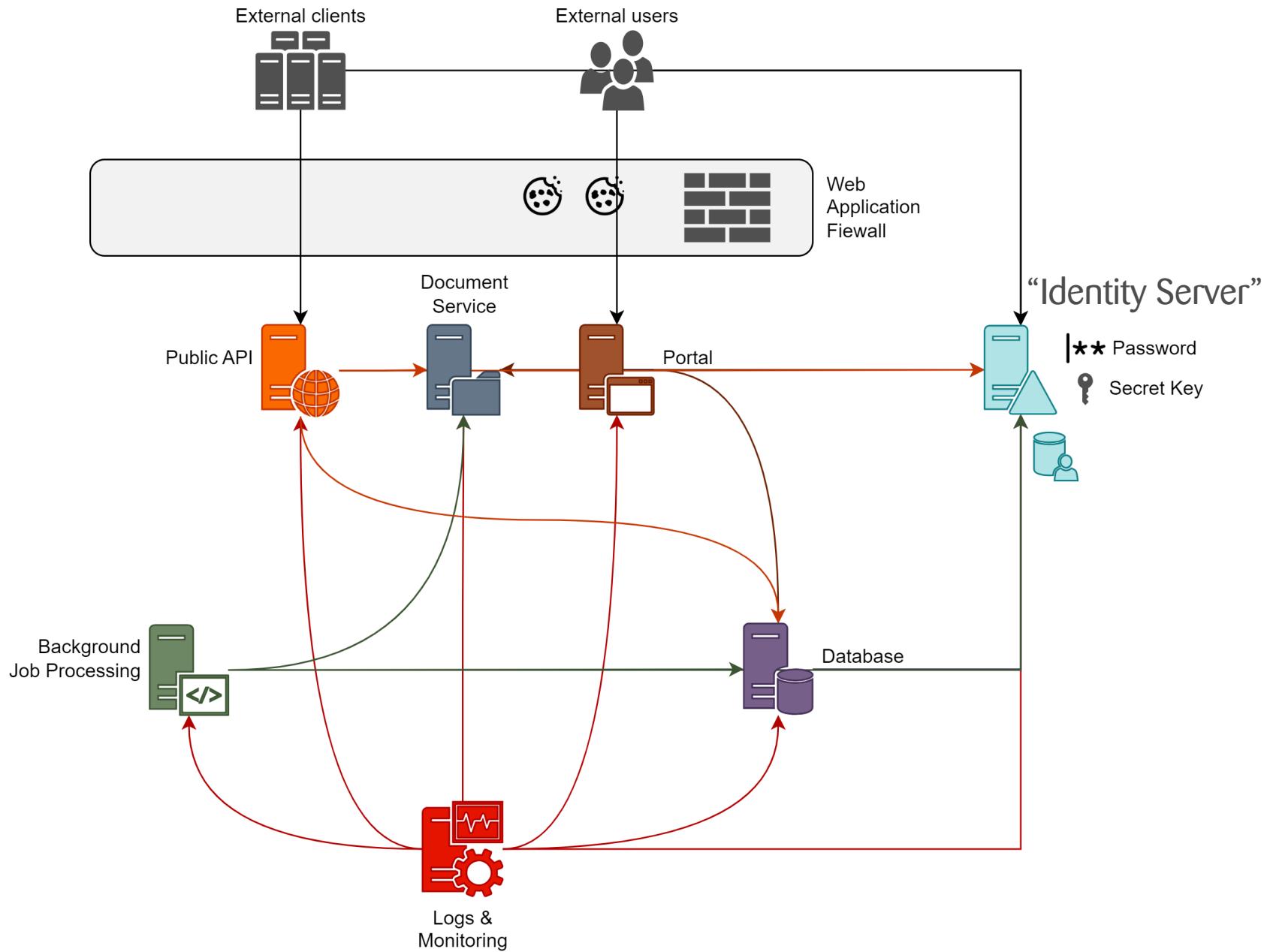
It has to know them in
order to trust them.



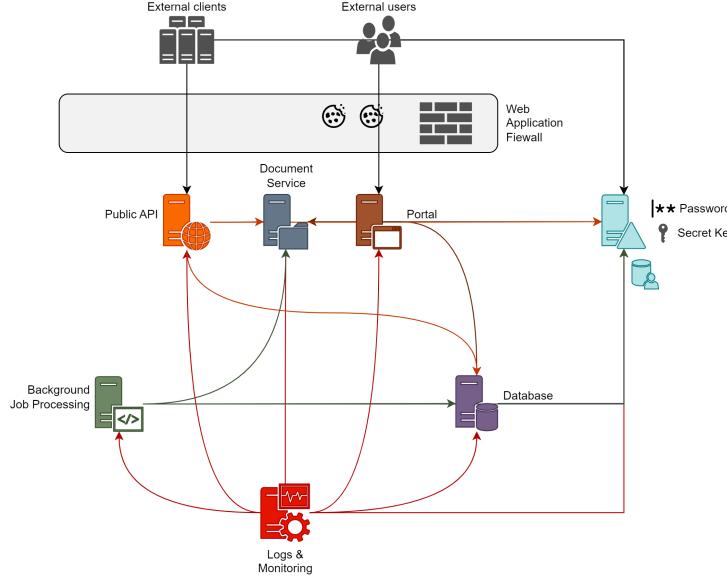
Single responsibility

Let's introduce a **single component** for authentication.

You now have a **single trust**.



Single sign-on (SSO)



SSO is an authentication scheme that allows a user or server to **sign-on once** and then with a **single identity** access many software systems.

It can be achieved with OAuth, Kerberos, SAML.

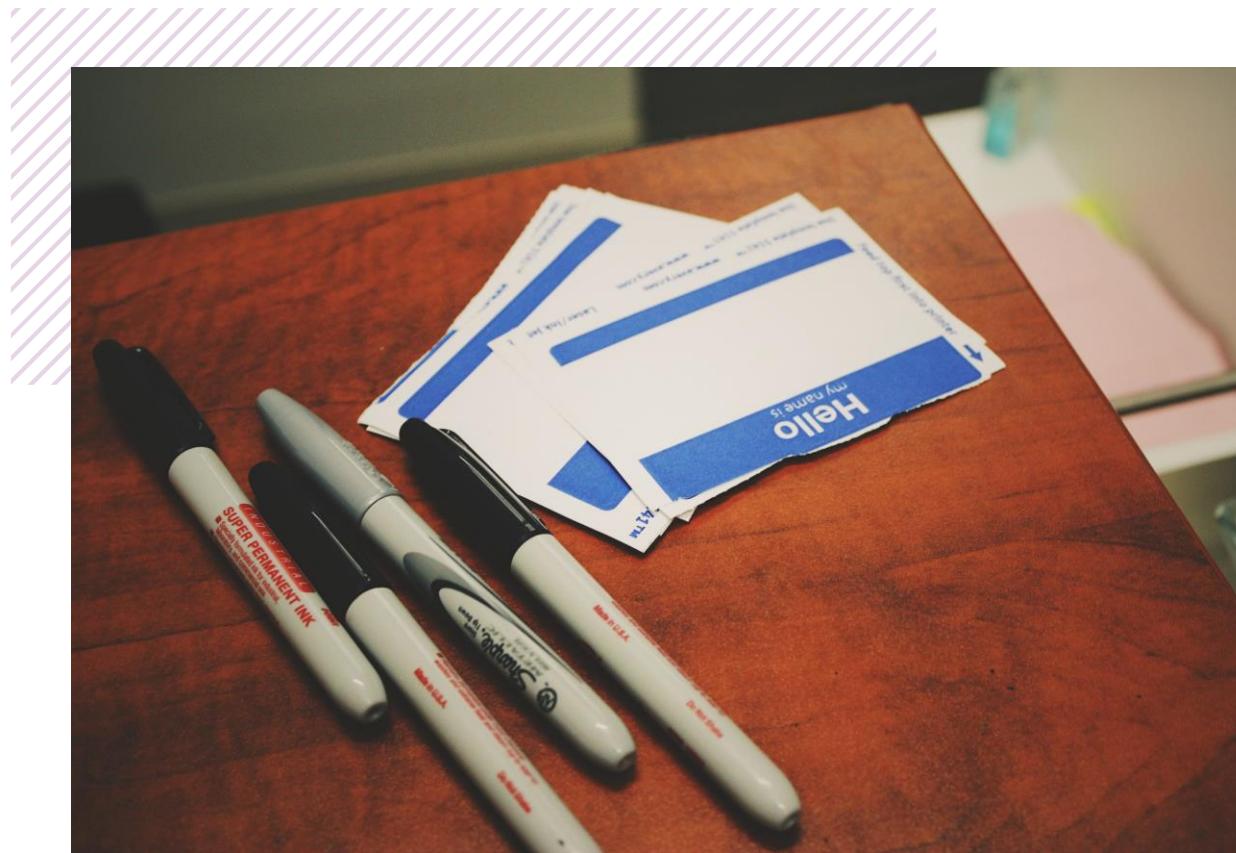
Example: I login to Google account and then use this to get authorized in other services.

Comparison of SSO protocols

For application-level SSO of web and mobile use OIDC

Criteria	OAuth/OpenID Connect	SAML	Kerberos (also with LDAP SASL Bind)
Description	Identity layer on top of OAuth 2.0 that verifies end-user identity.	Allows identity providers to pass authorization credentials to service providers.	Network authentication protocol using secret-key cryptography.
Best Use Case	Modern web and mobile apps, "login with" style federated identity.	Enterprise and B2B scenarios with Single Sign-On across multiple services.	Controlled corporate or campus environments for network-level authentication.
Data Format	JSON	XML	Binary
Commonly Used In	Web and mobile applications	Enterprise web applications	Windows Active Directory, Unix networks
Cons	Not suitable for network-level authentication.	More complex to implement, XML makes it less suitable for mobile and web apps.	Complexity in managing and setting up Kerberos servers, limited to controlled environments.

02



Learn the names

Terminology

Identity

OpenID Connect (OIDC)

JWT

Authentication

OAuth

Claims

Authorization

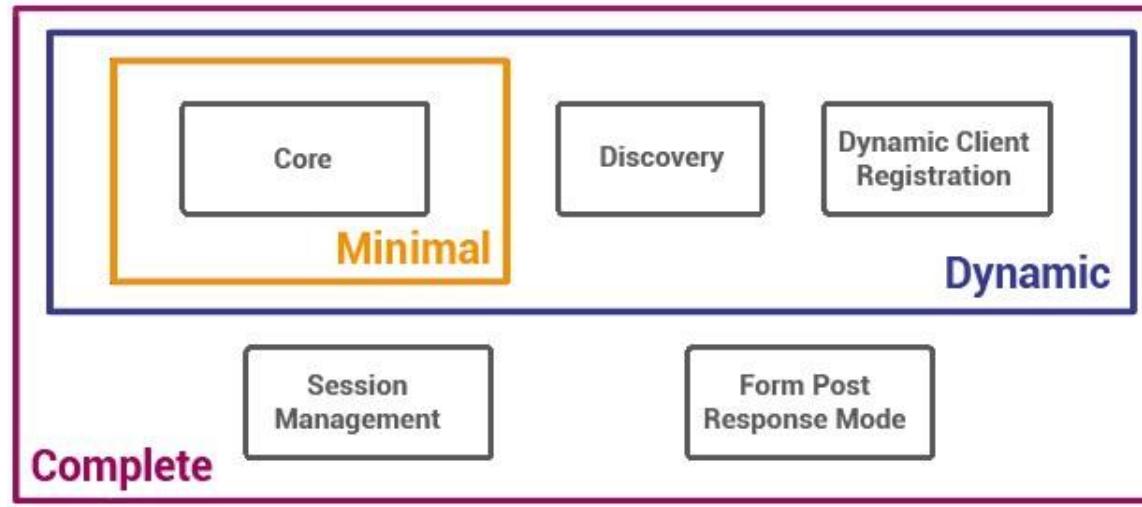
Scopes

Single sign-on (SSO)

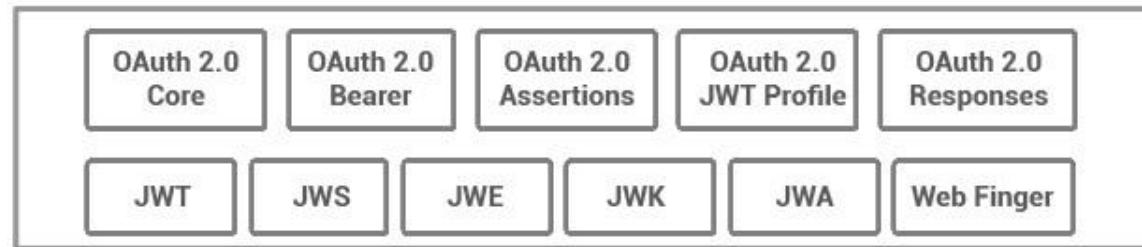
Bearer Token

OAuth VS OpenID Connect

OpenID Connect Protocol Suite



Underpinnings



OpenID Connect solves two problems in a unified way

Identity and Access Management (IAM)

Authentication

is the act of validating that users are whom they claim to be. It establishes their **Identity**.

Not valid → Status Code: 401



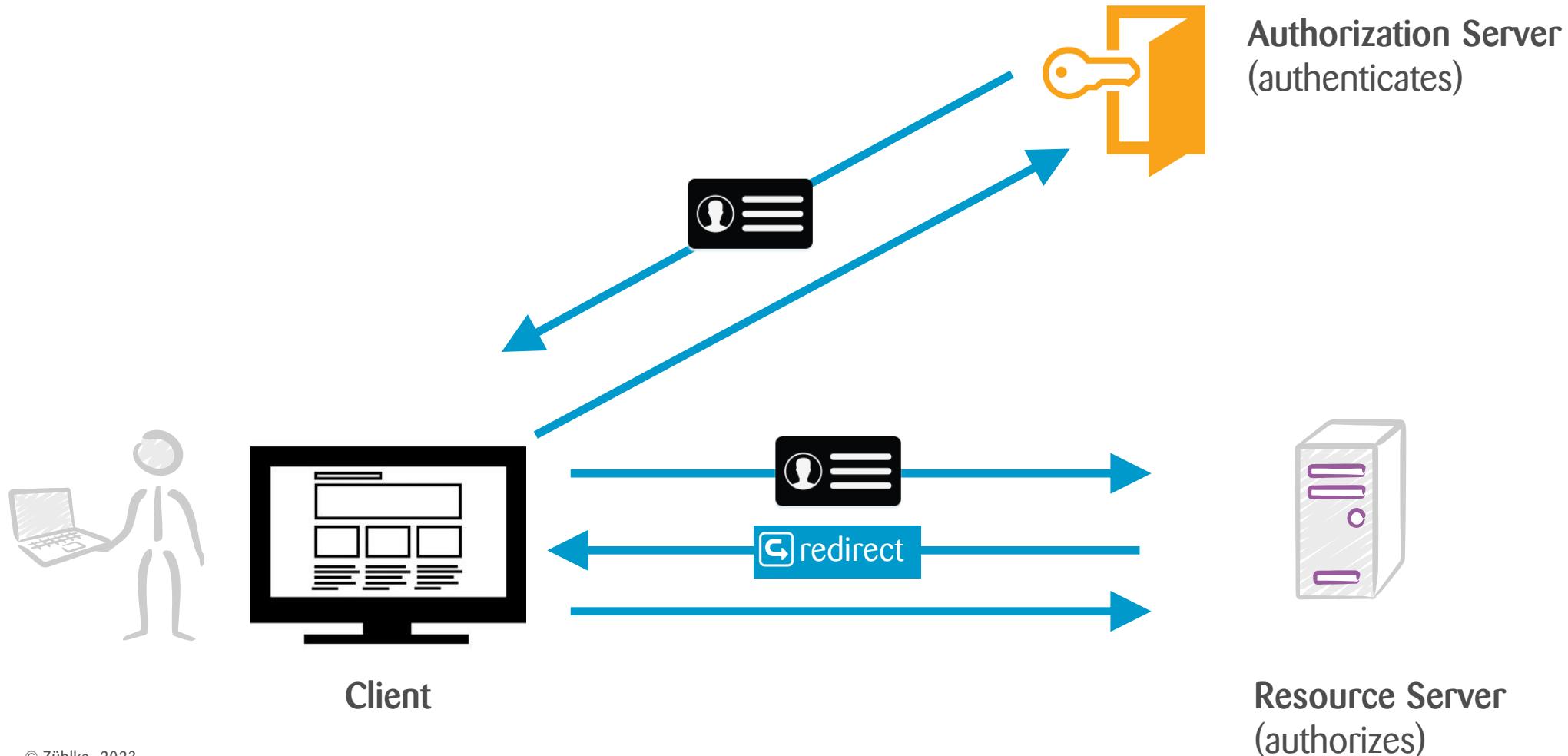
Authorization (Access Management)

is the process of determining if the user has **permission** to access a specific resource or function.

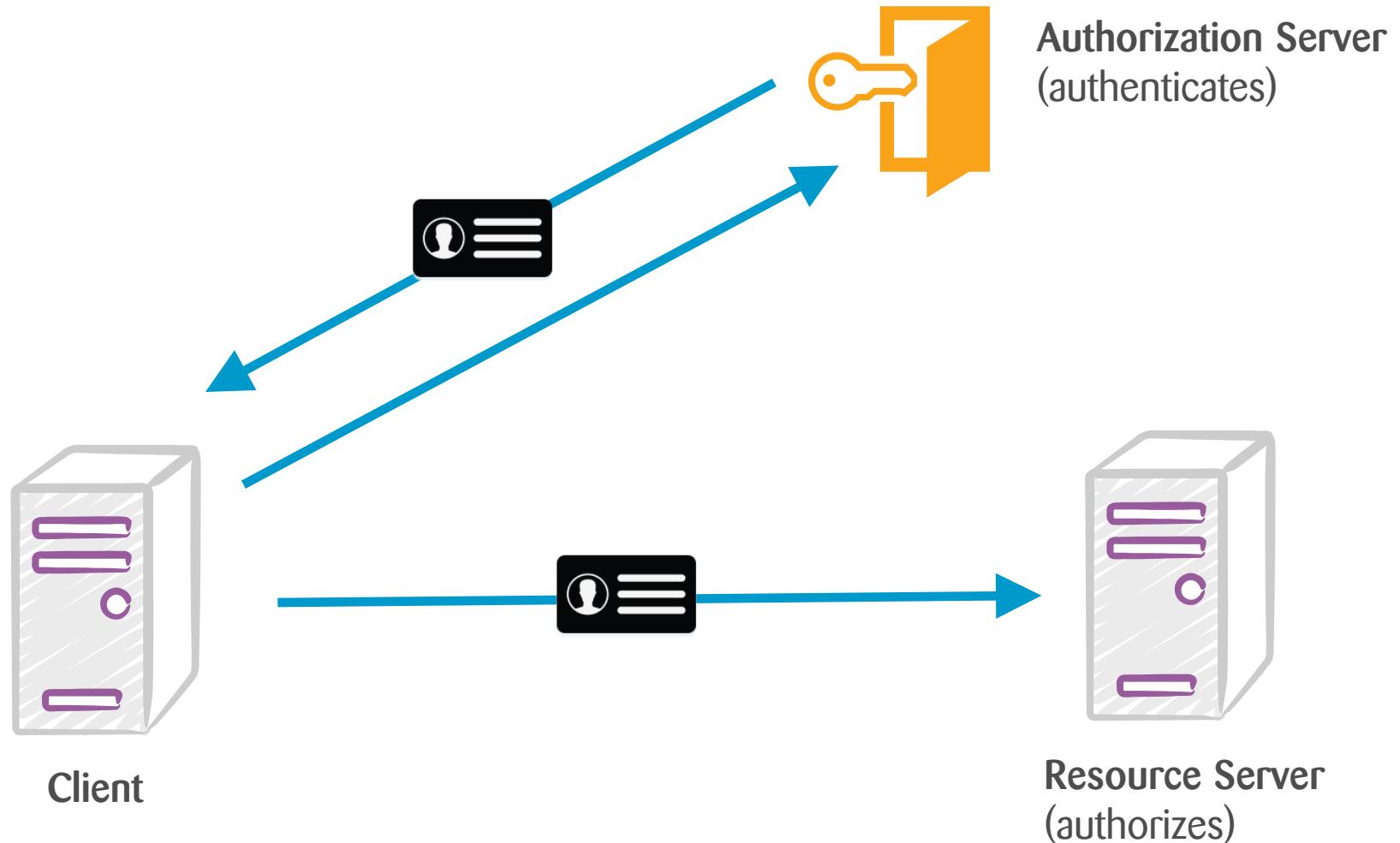
Not permitted → Status Code: 403



How does it work? Concept with a human.



How does it work? Concept without a human (just machines).

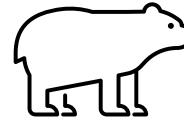


Bearer Token

Not



but



A **Bearer Token** is an opaque string, not intended to have any meaning to clients using it.

Some tokens are a short string of hexadecimal characters, others may be structured such as [JSON Web Tokens](#).

```
▼ Request Headers    view source
Accept: application/json, text/plain, /*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ
cm5hbWUiOiIjZHNtaXR0NzIiLCJuYW1lIjoiQ0QgU21pdGgiLCJlbWFpbCI6ImNkc21p
3MiOijCdWRnZXRpdmI0eSIsImF1ZCI6IjNBMERBQkE1LTc2MEItRTcxMS04RjM5LTc4M
JmIjoxNTEyODc1MjgxQ.6x0fiLKn45kf9r14SUX20U1oB9ggL4ZeKiPHuUiRPCY
Connection: keep-alive
```



Bearer Token

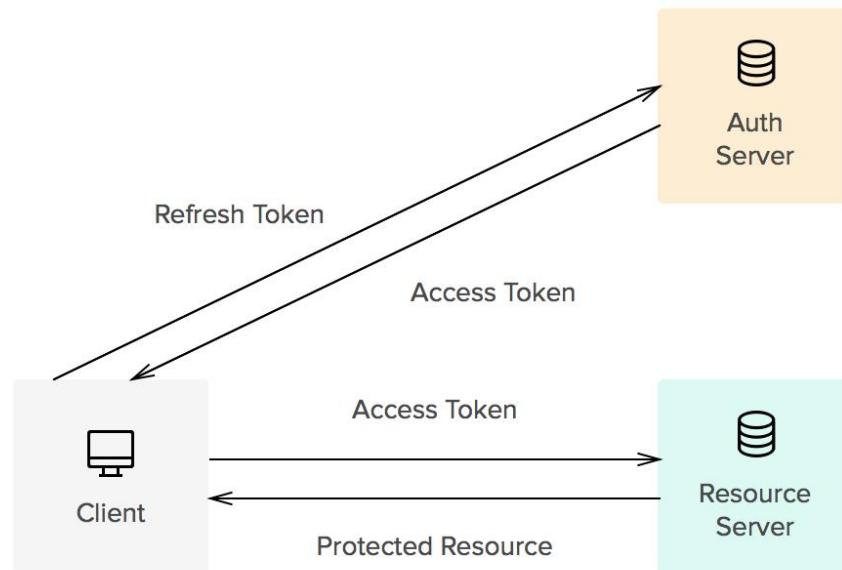
OpenId Connect token recognizes three types

- **access_tokens** (used by resource servers to **authorize or deny requests**)
- **id_token** (used by client applications to **identify users**)
- **refresh_token** (used by client applications to **get new token**)



Refresh token

Refresh tokens carry the information necessary to get a new access token.



ID/Access	Refresh
Short lived	Long lived
Used for auth	Used for getting new access token
Expires after period of inactivity	Lives during whole session
Storage less secure than Refresh	Storage more secure than Refresh

JSON Web Tokens (JWT)

JSON Web Token (RFC 7519) is a way to encode claims in a JSON document that is then signed (usually by a certificate private key). It gives information to the Resource Server about the Client.

The screenshot shows a browser window for jwt.io. The URL bar says "jwt.io". The page title is "JSON Web Tokens - jwt.io". The user is signed in as "Ryan". The algorithm selected is "HS256".
Encoded:
The encoded token is:
`eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpxVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaaG4gRG9
lIiwiYWRtaW4iOnRydW9.TJVA95
OrM7E2cBab30RMHrHDcEfxjoYZge
FONFh7HgQ`
Decoded:
HEADER:
`{
 "alg": "HS256",
 "typ": "JWT"
}`
PAYOUT:
`{
 "sub": "1234567890",
 "name": "John Doe",
 "admin": true
}`
VERIFY SIGNATURE
`HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
secret
)` secret base64 encoded
Signature Verified

Claims

Claims

Claims are name/value pairs that contain information about a user, as well meta-information about the OIDC service. Official definition from spec: “piece of information asserted about an Entity.”

The "sub" (subject) claim identifies the identity that is the subject of the JWT.

```
{  
  "family_name": "Silverman",  
  "given_name": "Micah",  
  "locale": "en-US",  
  "name": "Micah Silverman",  
  "preferred_username": "micah.silverman@okta.com",  
  "sub": "msilver",  
  "updated_at": 1490198843,  
  "zoneinfo": "America/Los_Angeles"  
}
```

Scopes

Scopes are space-separated lists of identifiers used to specify what claims are being requested by the **client**. Identity Server will grant them if this **client** is allowed to know them.

scope	purpose
profile	requests access to default profile claims
email	requests access to email and email_verified claims
address	requests access to address claim
phone	requests access to phone_number and phone_number_verified claims



The default **profile** claims are:

- name
- family_name
- given_name
- middle_name
- nickname
- preferred_username
- profile
- picture
- website
- gender
- birthdate
- zoneinfo
- locale
- updated_at

03



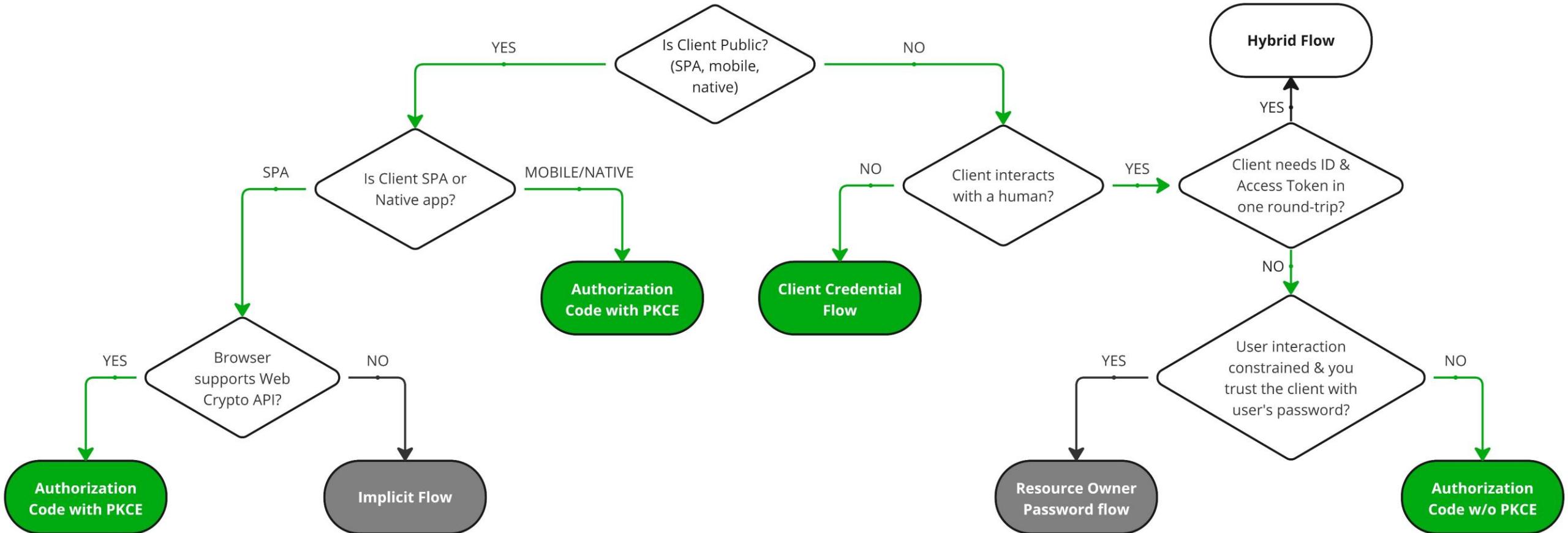
Going with the flow

How to choose an OIDC Flow / Grant

Flows

The OAuth 2.0 supports several different **flows** (or **grants**). Flows are ways of retrieving an Access Token.

Which to use depends mostly on application type, but other parameters as well: level of trust for the client, or the experience you want your users to have...



04



Mind the gap

Implementing and common pitfalls

What do you need for implementation?

A library per Actor



* All 3 components can use different technologies.
OIDC is a specification - **technology agnostic**.

Implementation libraries

OpenID Connect keeps a list of all libraries:

<https://openid.net/developers/libraries/>

Configuring

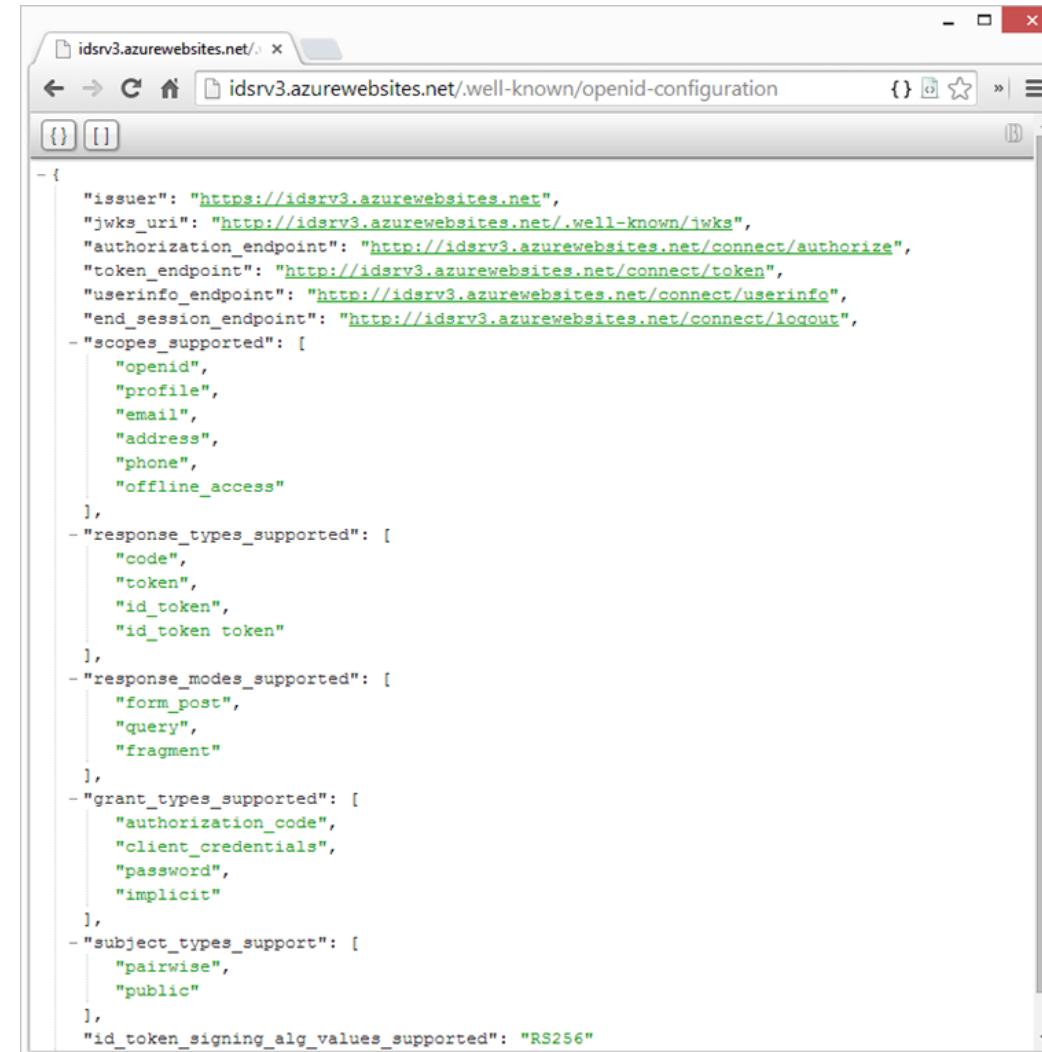
Configuration depends on the Flow/Grant

Read the chosen library docs.

Resources Server and Client Server can auto configure using the Discovery endpoint that the Authorization Server will provide

Example: `https://server.com/.well-known/openid-configuration`

.well-known/openid-configuration



The screenshot shows a browser window with the URL `idsrv3.azurewebsites.net/.well-known/openid-configuration`. The page displays a JSON object representing the OpenID Configuration document. The JSON structure includes fields such as issuer, jwks_uri, authorization_endpoint, token_endpoint, userinfo_endpoint, end_session_endpoint, scopes_supported, response_types_supported, response_modes_supported, grant_types_supported, subject_types_supported, and id_token_signing_alg_values_supported.

```
{  
  "issuer": "https://idsrv3.azurewebsites.net",  
  "jwks_uri": "http://idsrv3.azurewebsites.net/.well-known/jwks",  
  "authorization_endpoint": "http://idsrv3.azurewebsites.net/connect/authorize",  
  "token_endpoint": "http://idsrv3.azurewebsites.net/connect/token",  
  "userinfo_endpoint": "http://idsrv3.azurewebsites.net/connect/userinfo",  
  "end_session_endpoint": "http://idsrv3.azurewebsites.net/connect/logout",  
  "scopes_supported": [  
    "openid",  
    "profile",  
    "email",  
    "address",  
    "phone",  
    "offline_access"  
,  
  "response_types_supported": [  
    "code",  
    "token",  
    "id_token",  
    "id_token token"  
,  
  "response_modes_supported": [  
    "form_post",  
    "query",  
    "fragment"  
,  
  "grant_types_supported": [  
    "authorization_code",  
    "client_credentials",  
    "password",  
    "implicit"  
,  
  "subject_types_supported": [  
    "pairwise",  
    "public"  
,  
  "id_token_signing_alg_values_supported": "RS256"
```

Common mistakes 1/2

...and how to avoid them

Learn the HTTP basics

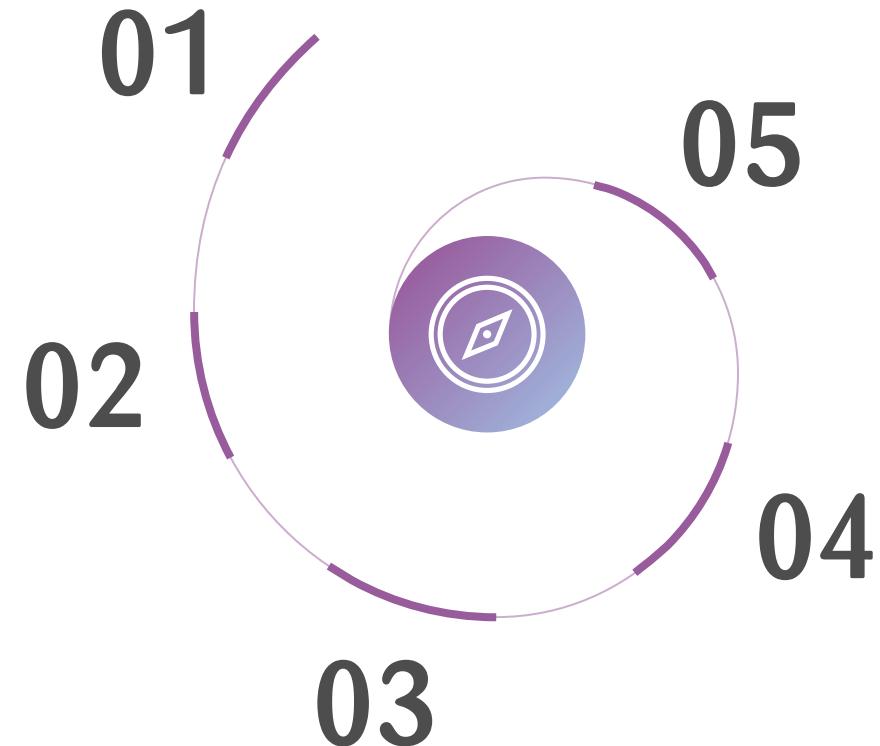
Understand HTTP request/ response exchange and headers vs body.

Understand the flow

Once you selected the flow – learn the HTTP request/response sequence diagram of the flow.

Understand the role of Actors

Know the difference between Client, Resource and Authorization Server.



Handle ID/access token expiry

Check if ID/access token is expired. If yes, get new by using refresh token.

⚠ Match configuration between actors **exactly**

redirect_url: [https://myresource
server.com/resource/](https://myresourceserver.com/resource/)
scopes: profile

Common mistakes 2/2

...and how to avoid them

Handle refresh token expiry

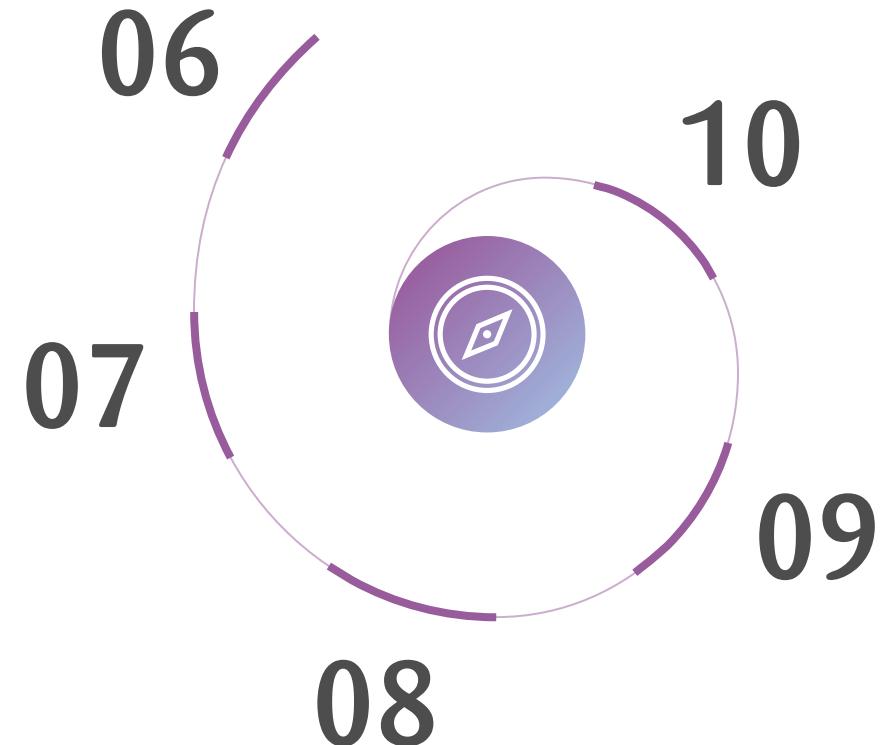
Check if refresh token expired.
If yes, re-authenticate user or show
“Session expired, click to login”.

Provide logout

In case user doesn't use their
own device to login.

Logging is important!

OIDC is complicated. Logs will save
you a lot of time when debugging.



Be aware of intermediate actors

When debugging, be aware of
firewalls, proxies etc.

Each environment is different

Dev usually: mock Auth Server, mock
components, missing firewall.

What works in Dev might not work in
Production!