Introduzione a OAuth 2.0 e OpenID Connect

Enrico Vianello, Federica Agostini, Roberta Miccoli

“Panoramica su OAuth2/OpenID Connect e sue applicazioni tramite il servizio INDIGO IAM”
12-14 Maggio 2025, Frascati (Roma)

Istituto Nazionale di Fisica Nucleare
Laboratori Nazionali di Frascati

Part 1: The protocols

a step by step overview of the protocol concepts

- Introduction
- OAuth 2.0 + OpenID Connect
- Authorization Code Flow
- The well-known endpoints
- JSON Web Tokens (JWT)
- PKCE
- Token based Authorization
- Long-Term access with Refresh Tokens
- Client Credentials Flow
- Device Code Flow
- Token Exchange Flow
- Dynamic Client Registration
- OAuth 2.1

Introduction

Once upon a login ...

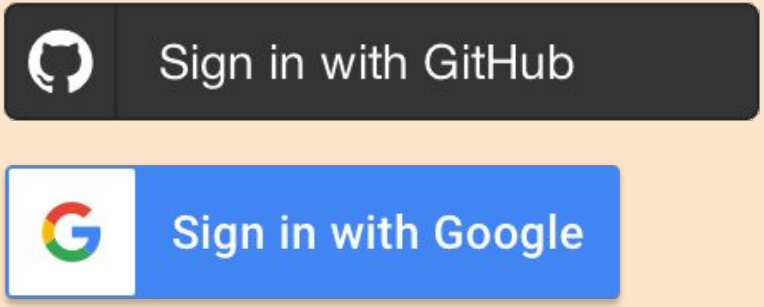
You had an email,
maybe a Facebook.
Life was good.

Then ...

you needed an account for
***ordering pizza**,*
***buying socks**,*
***posting vacation selfies**,*
***renting cars**,*
***playing yet another game**, …*

… and suddenly,
you had **37 usernames**
and at least **4 variations of the same password**
one of which might be “Password123!”

the hero we needed



The problem we're solving

A user has access to a resource
and wants to allow a third party to have the same access
without sharing their credentials/passwords

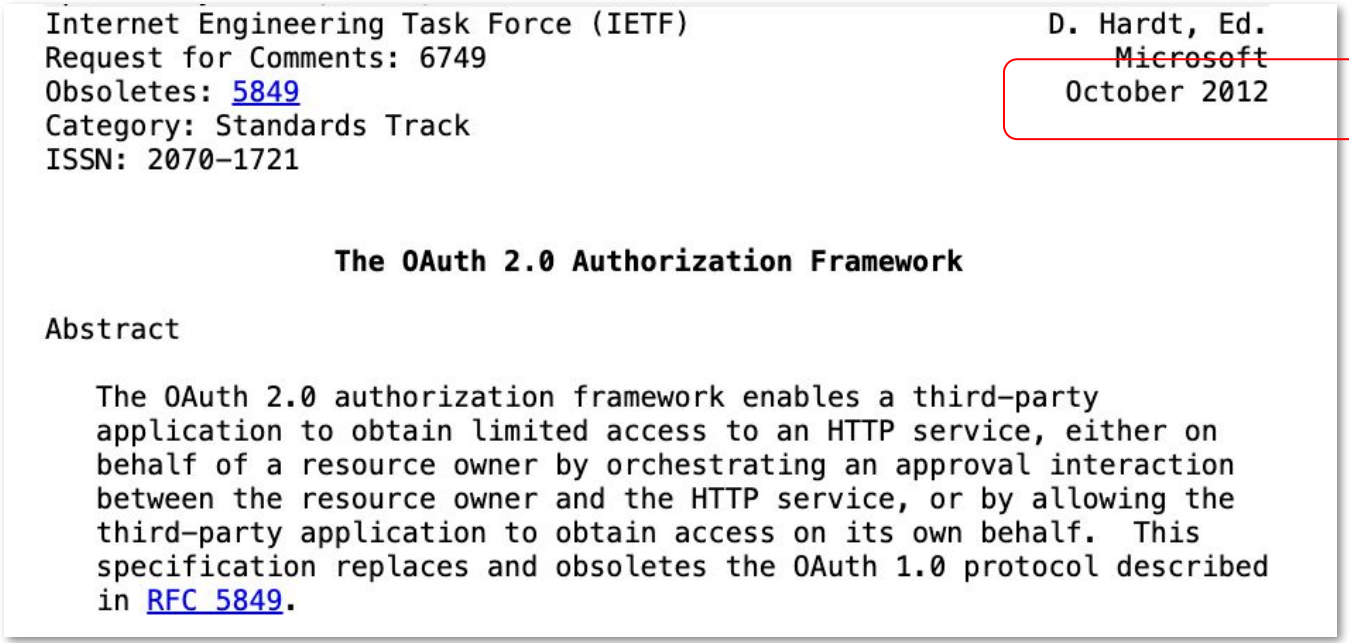
In the "old days" this could be solved by the third party saving the username and password of the user, which would allow the third party to impersonate the user when accessing the resource.

But that is less than optimal because of two points:

- if the user changes the password, all third parties lose access
- there's no guarantee that an organization/service will keep your credentials safe, or guarantee their service won't access more of your personal information than necessary

OAuth 2.0 + OpenID Connect

Open **Auth**orization Framework



Internet Engineering Task Force (IETF)
Request for Comments: 6749
Obsoletes: 5849
Category: Standards Track
ISSN: 2070-1721

D. Hardt, Ed.
Microsoft
October 2012

The OAuth 2.0 Authorization Framework

Abstract

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

it's a long time…

RFC 6749 - https://www.rfc-editor.org/rfc/rfc6749

Internet Engineering Task Force (IETF)
Request for Comments: 8252
BCP: 212
Updates: 6749
Category: Best Current Practice
ISSN: 2070-1721

OAuth 2.0 for Native Apps

Abstract

OAuth 2.0 authorization reques
through external user-agents,
specification details the secu
the case and how native apps
this best practice.

Internet Engineering Task Force (IETF)
Request for Comments: 7662
Category: Standards Track
ISSN: 2070-1721

OAuth 2.0 Token Introspection

Abstract

This specification defines a
an OAuth 2.0 authorization
OAuth 2.0 token and to dete
OAuth 2.0 deployments can us
the authorization context of
to the protected resource.

Inter
Reques
Categ
ISSN: 2070-1721

OAuth 2.0 Token Revocatio

stract

This document proposes an additional endpoint for OAuth authorization
servers, which allows clients to notify the authorization server that
a previously obtained refresh or access token is no longer needed.
This allows the authorization server to clean up security
redentials. A revocation request will invalidate the actual token
nd, if applicable, other tokens based on the same authorization
ant.

W. Dennis
Googl
J. Bradl
Ping Ident
October

Request for Comments: 8693
Category: Standards Track
ISSN: 2070-1721

January 2020

M. Jones
Microsoft
A. Nadalin
Microsoft
B. Campbell, Ed.
Ping Identity
J. Bradley
Yubico
C. Mortimore
Visa

OAuth 2.0 Token Exchange

Abstract

This specification defines a protocol for an HTTP- and JSON-based
Security Token Service (STS) by defining how to request and ob
security tokens from OAuth 2.0 authorization servers
security tokens employing impersonation and d

many updates have followed

Final

N. Sakimura
NAT.Consulting (was at NRI)
J. Bradley
Yubico (was at Ping Identity)
M. Jones
Self-Issued Consulting (was at
Microsoft)
B. de Medeiros
Google
C. Mortimore
Disney (was at Salesforce)
December 15, 2023

OpenID Connect Core 1.0 incorporating errata s

Abstract

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It enables Clients to verify the identity of the End-U
based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-Use
erable and REST-like manner.

ID Connect functionality: authentication built on top of OAuth 2.0 and the use of Claims to
escribes the security and privacy considerations for using OpenID Connect

Example

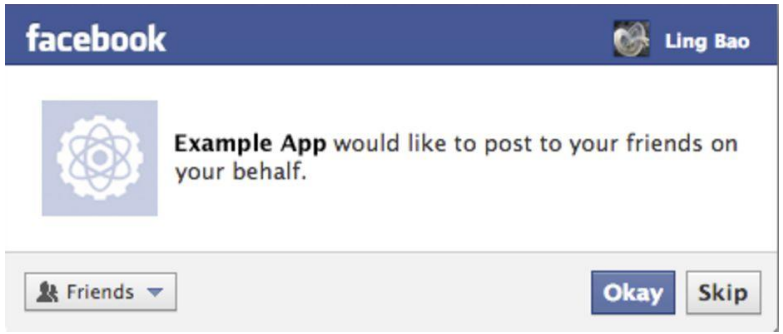
- You've probably run into a dialog saying something like this:

  " *Hey, this app is asking for access to your Facebook account, but won't publish posts on your behalf* "

  This is a very common pattern is OAuth

delegation of privileges

no shared passwords



OpenID Connect

OpenID Connect **extends OAuth** to provide a standard **identity layer**

- i.e. information about **who the logged user is** and **how it was authenticated**

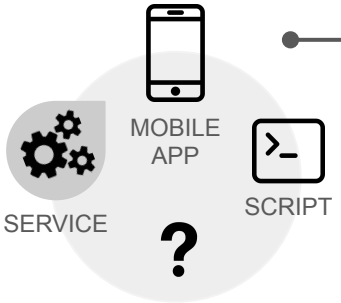
e.g. you could see OAuth 2.0 as the key of your car while OpenID Connect is your driving license

OIDC provides ability
to establish **login sessions** (Single-Sign On)

[OpenID Connect specification]

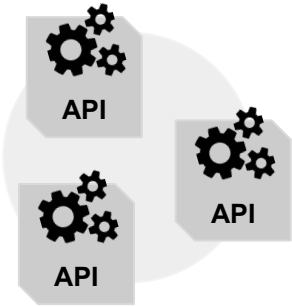
OAuth 2.0 / OpenID Connect in action

the TOKEN SERVICE that knowns users and clients and issues valid credentials/tokens



a CLIENT can be a lot of things, from a simple piece of software to a complex service

The USER

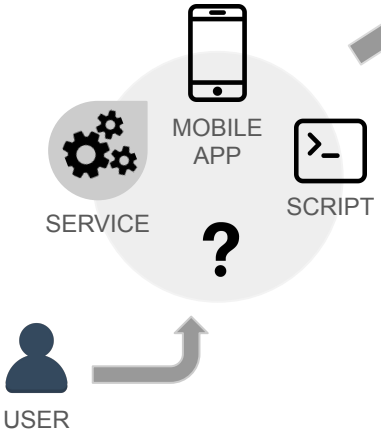


a RESOURCE the CLIENT wants to access
→ can be a generic API

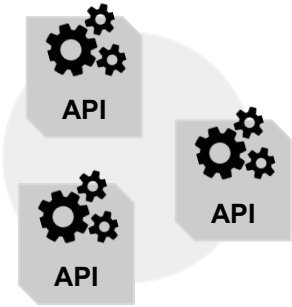
OAuth 2.0 / OpenID Connect in action

Authenticate the user for me?

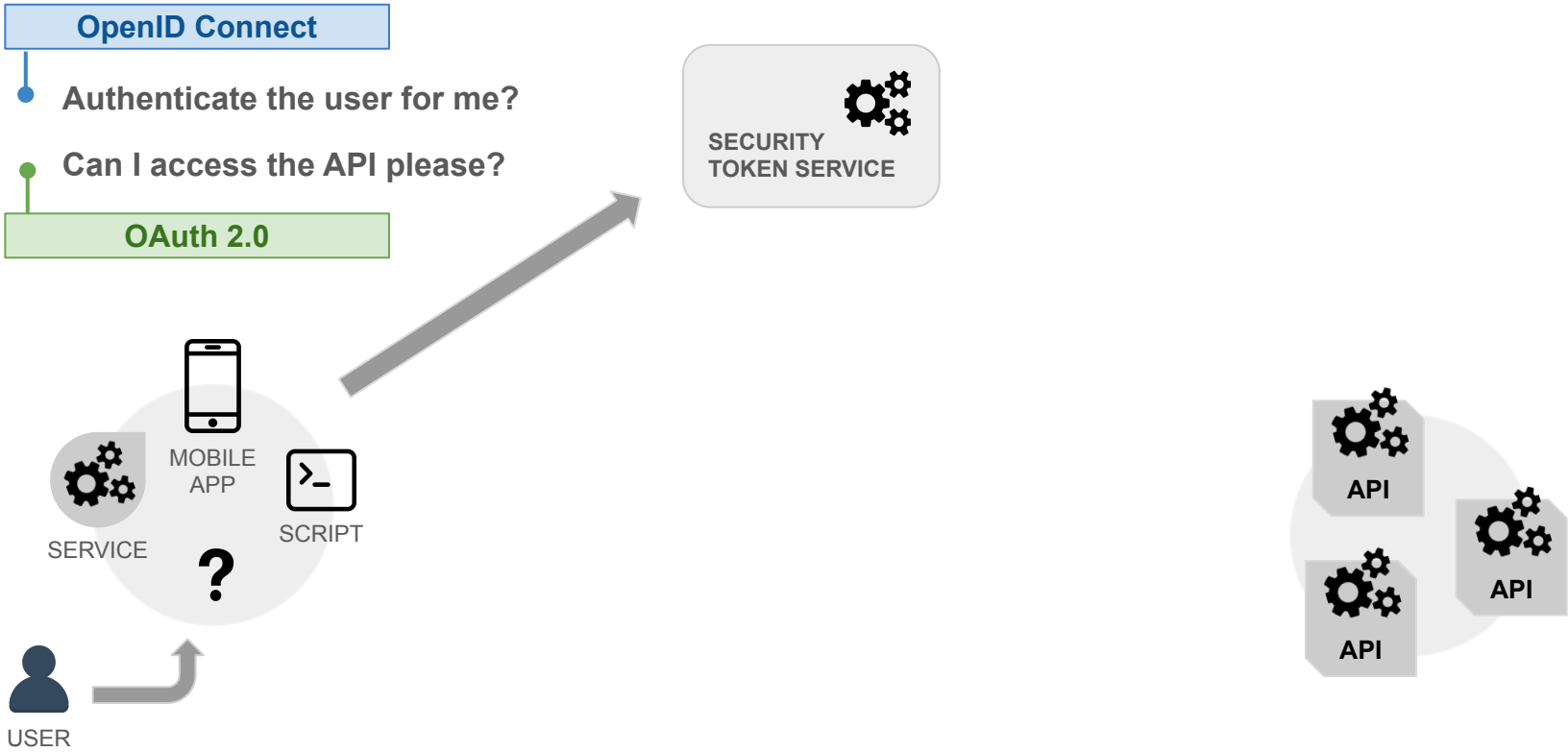
Can I access the API please?



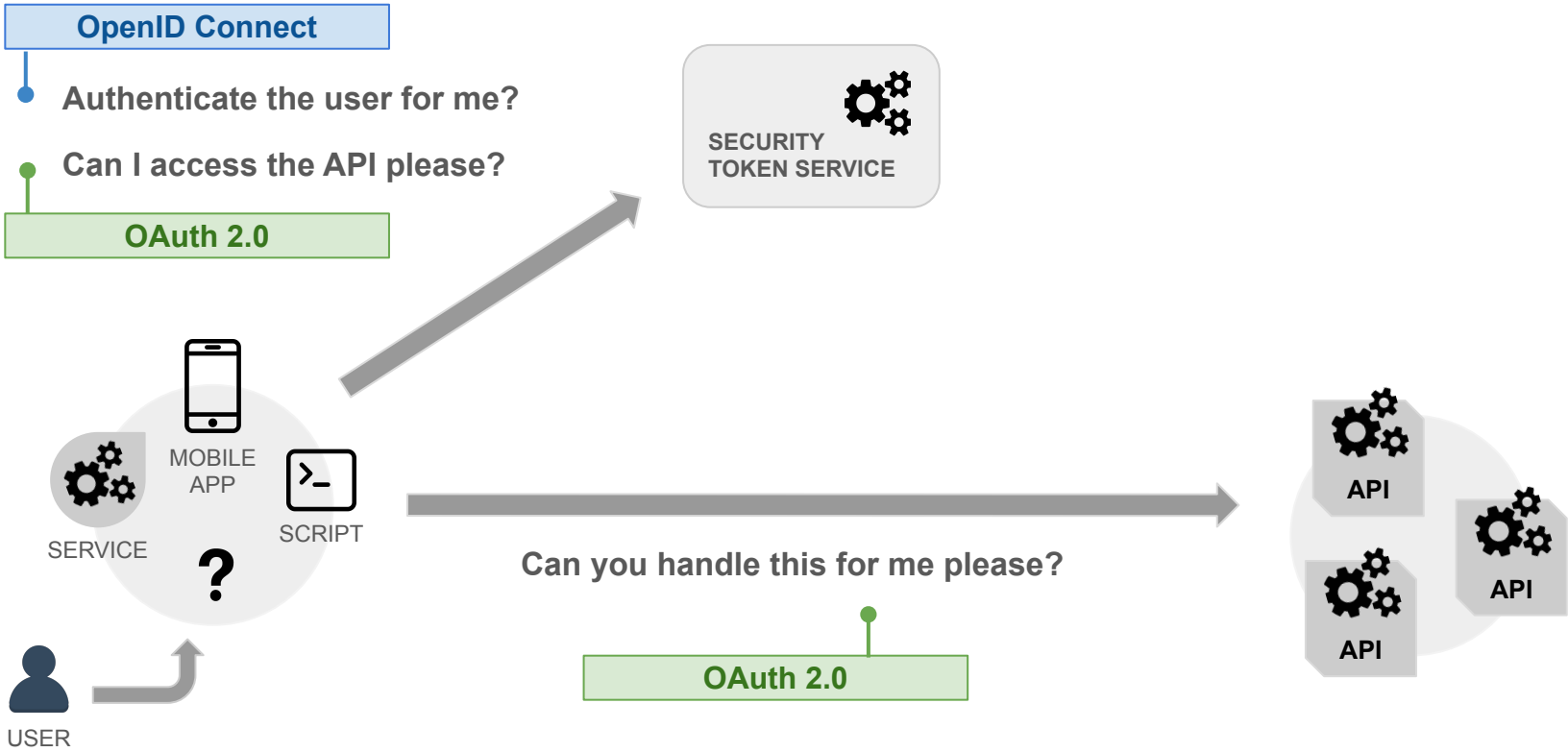
a CLIENT can ask to the Security Token Service things like …



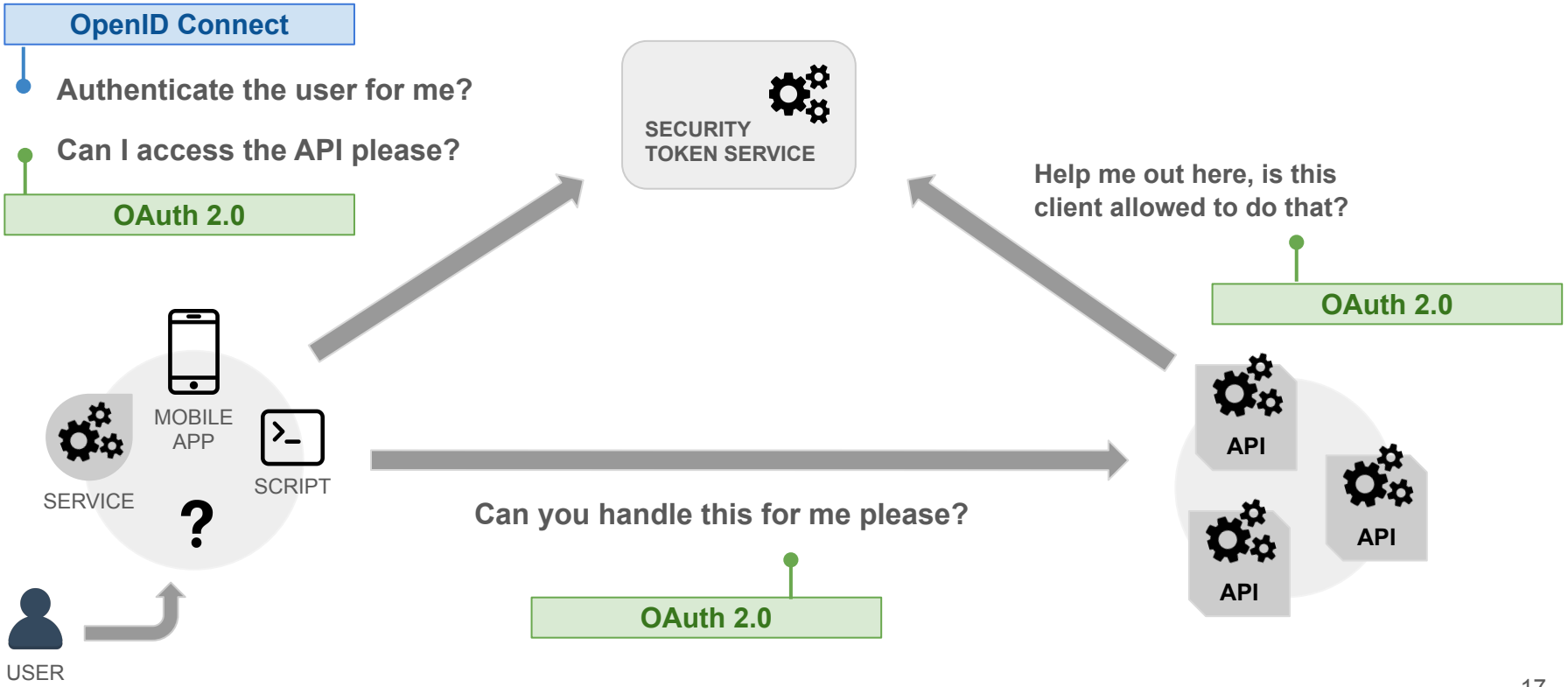
OAuth 2.0 / OpenID Connect in action



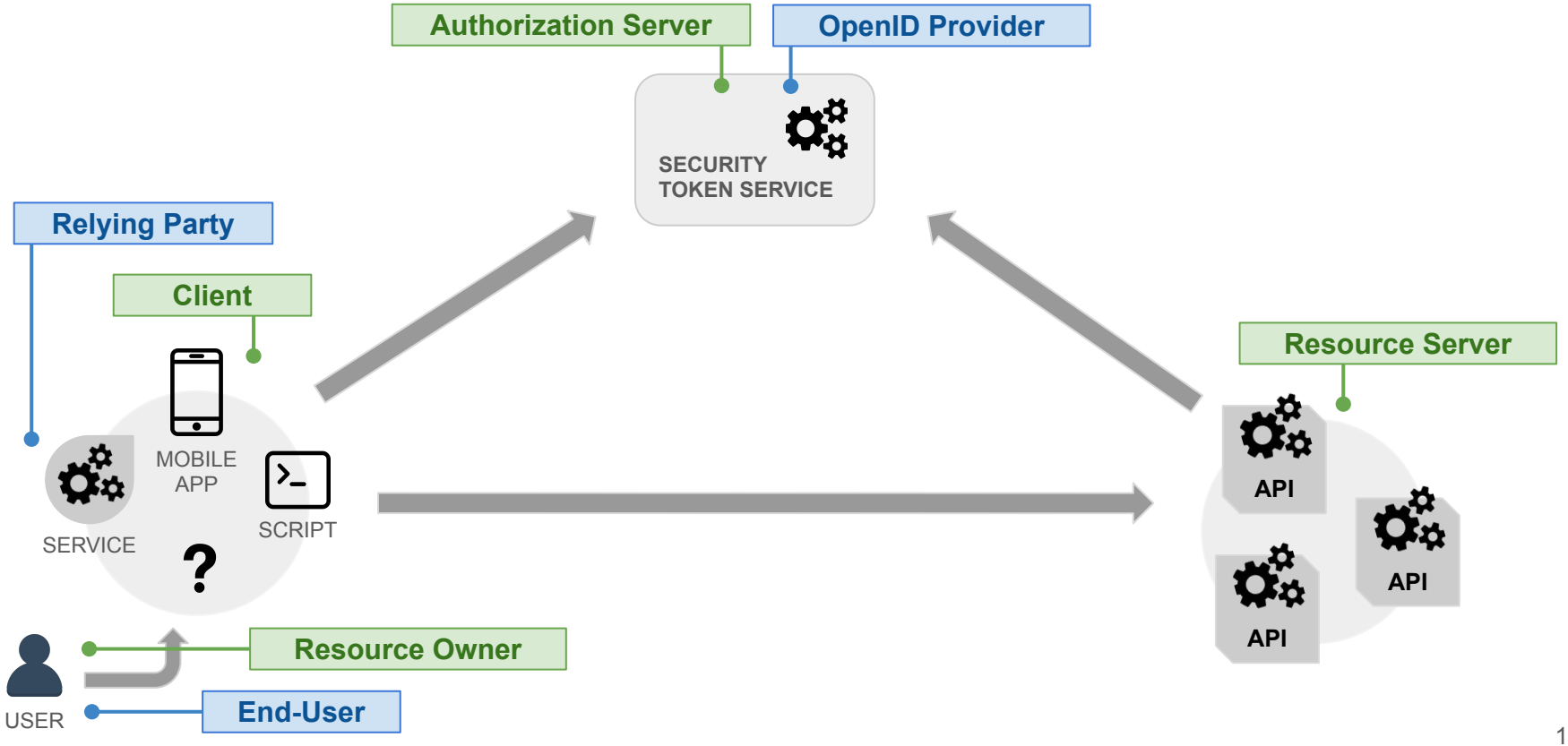
OAuth 2.0 / OpenID Connect in action



OAuth 2.0 / OpenID Connect in action



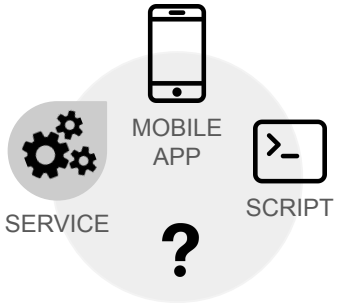
Terminology



OAuth2 / OpenID Connect roles/terminology (1)

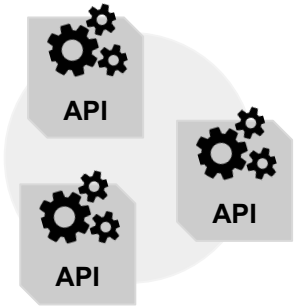
Resource Owner / **End-User**: A user that owns resources hosted at a service → *You, the owner of your identity, your data, and any actions that can be performed with your accounts*.

Client / **Relying-Party**: The application (or service, script, etc.) that wants to access data or perform actions **on behalf of the Resource Owner**.



OAuth2 / OpenID Connect roles/terminology (2)

Authorization Server / OpenID Provider: The application that knows the Resource Owner and the Client, where the Resource Owner already has an account and authenticates.
It issues tokens to Clients that can be used to access Resource Owner resources → **Resource Server**.

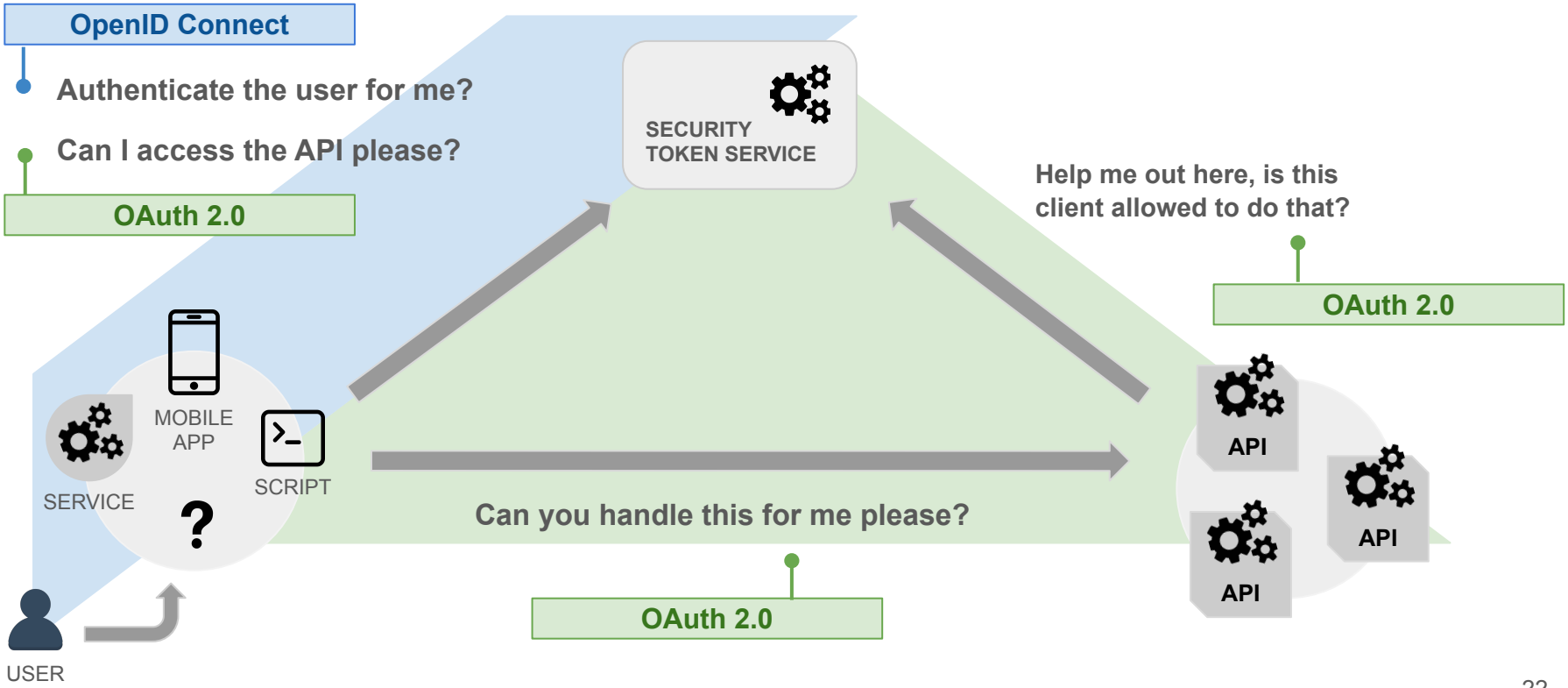


Resource Server: The Application Programming Interface (API) or service the Client wants to use on behalf of the Resource Owner.

Overview of use cases for OAuth and OIDC

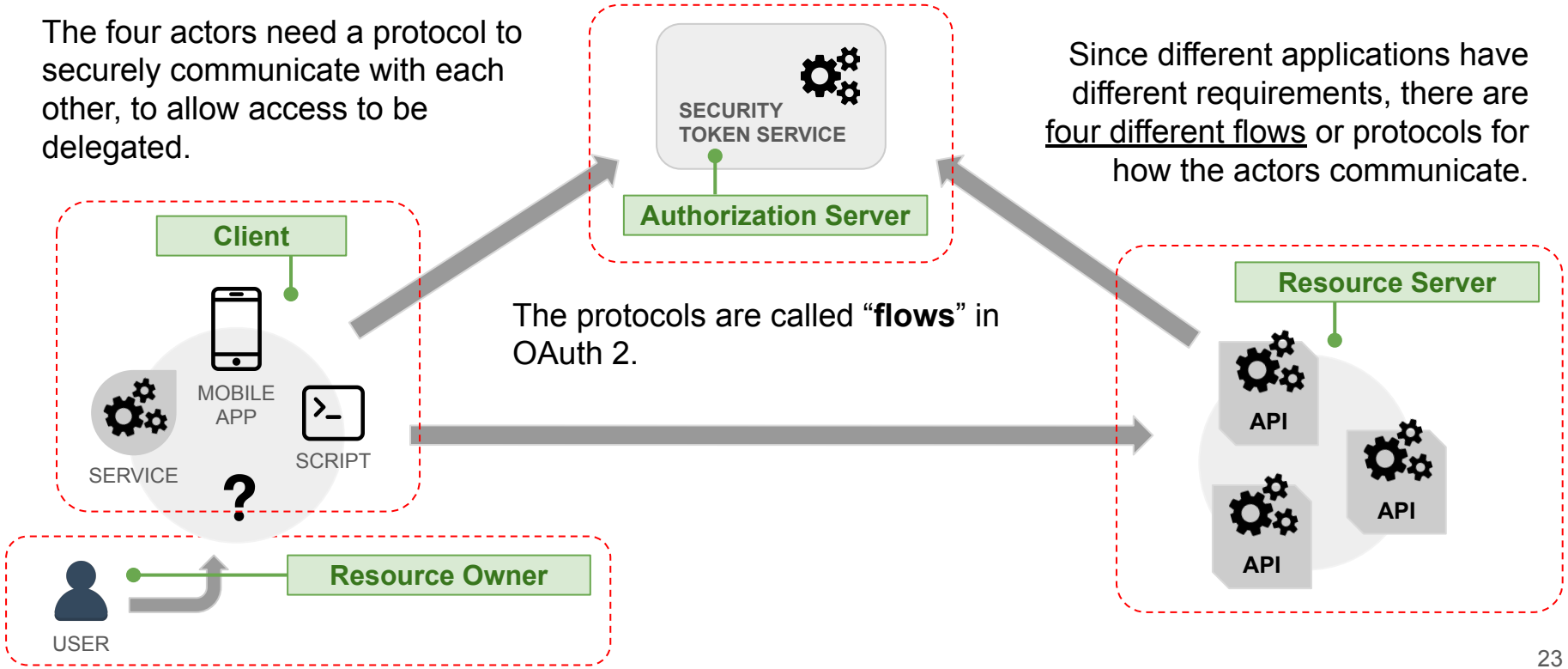
- An application wants to authenticate users using an external identity provider
  - e.g. delegating login to a different identity provider, social login or enterprise SSO
  - The client that wants to authenticate the user needs an **identity token**
  - **This scenario only uses OpenID Connect**
- An application wants to use an API on behalf of the user
  - e.g. accessing an API to retrieve all the contacts of a user
  - The client needs an **access token** to make requests to the API
  - **This scenario only uses OAuth**
- An application wants to authenticate users and access APIs on their behalf
  - e.g. a mobile app that authenticates the user and then retrieves all his contacts from API on his behalf
  - The client needs both an **identity token** and an **access token**
  - **This scenario combines OpenID Connect and OAuth**

Overview of use cases for OAuth and OIDC



How the parties communicate?

The four actors need a protocol to securely communicate with each other, to allow access to be delegated.



The protocols are called "**flows**" in OAuth 2.

Since different applications have different requirements, there are four different flows or protocols for how the actors communicate.

OAuth and OIDC flows

Authorization Grant Types

=

Authorization Flows

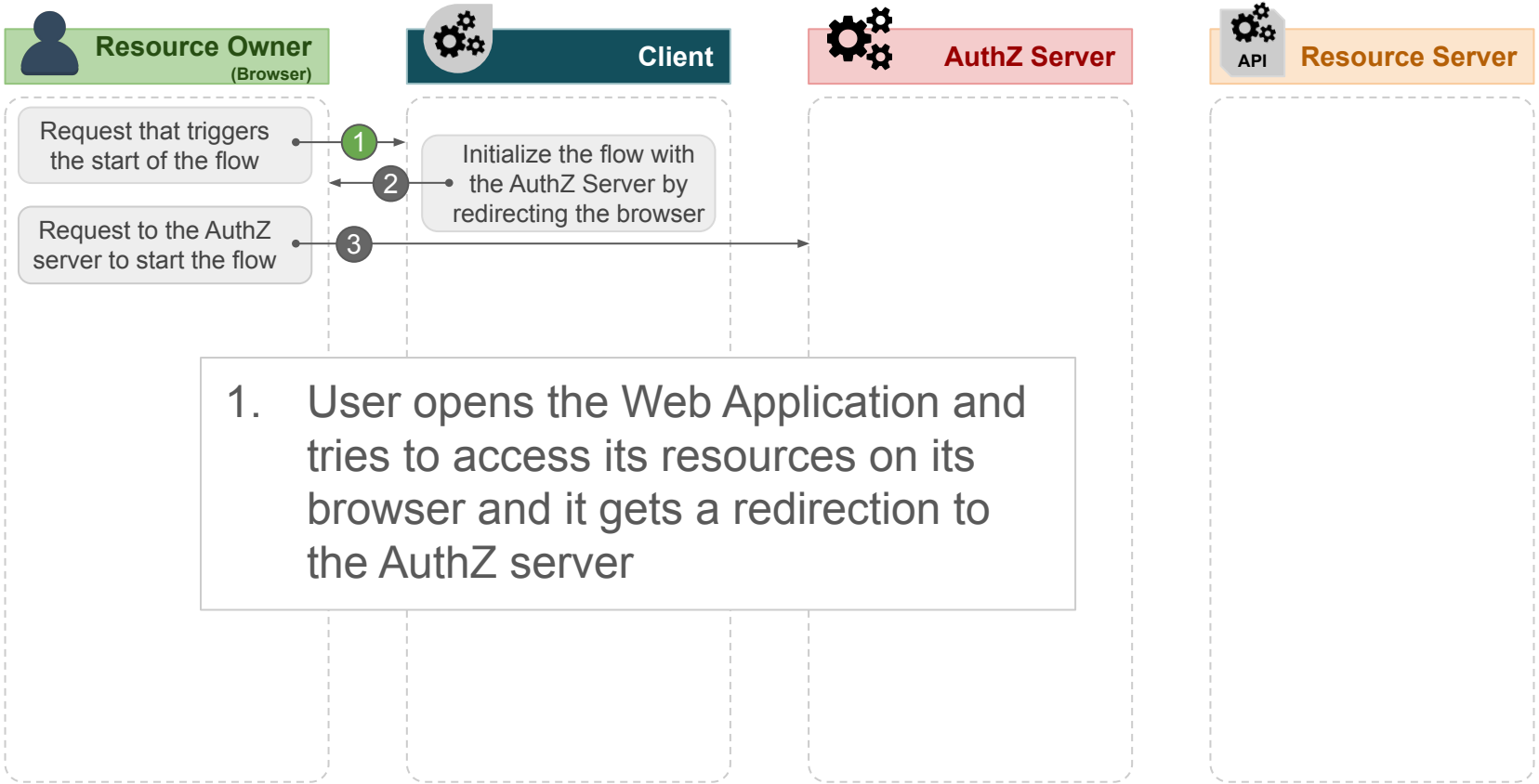
=

the set of steps a client uses to obtain one or more tokens from the authorization server, for the purpose of accessing a resource

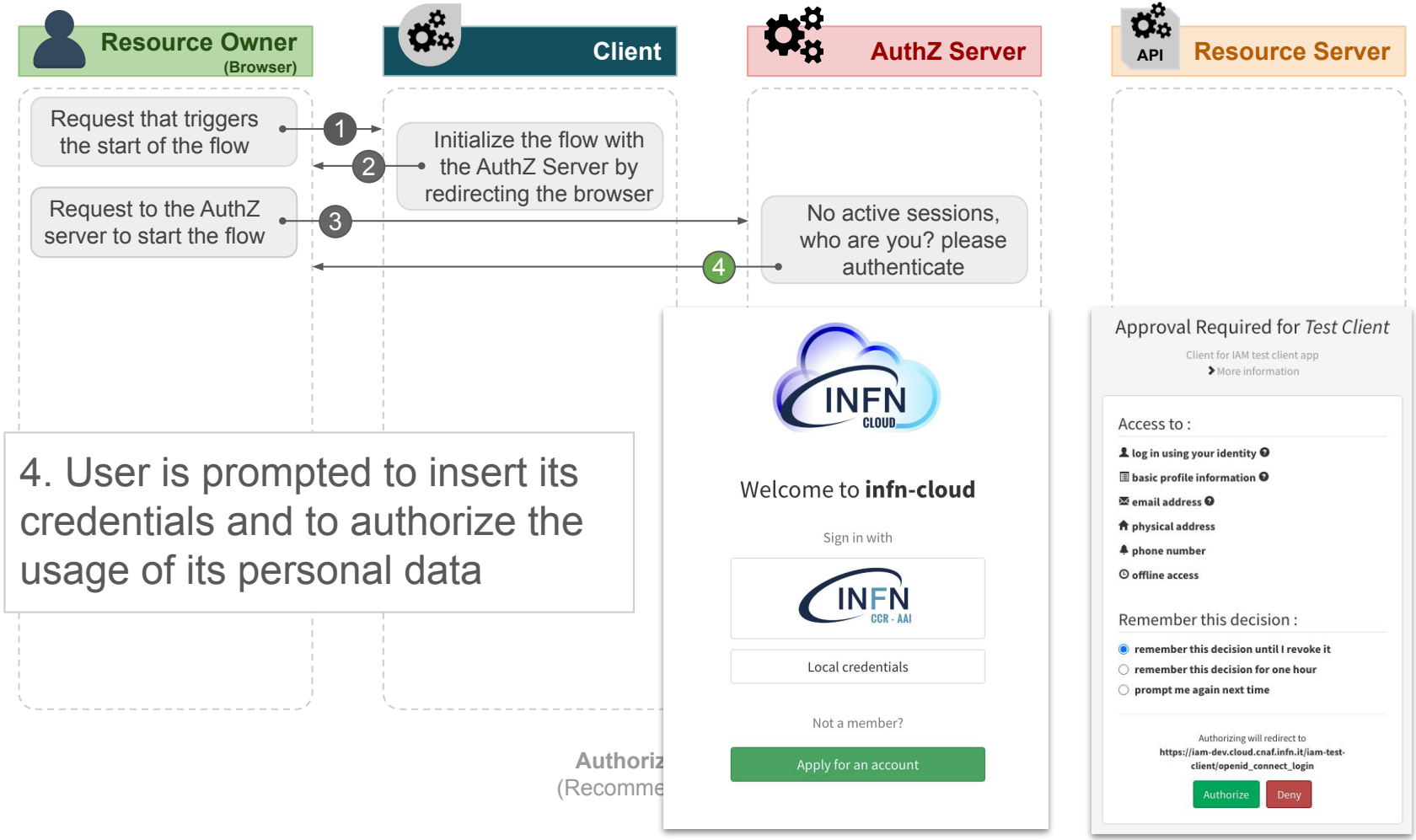
OAuth and OIDC flows

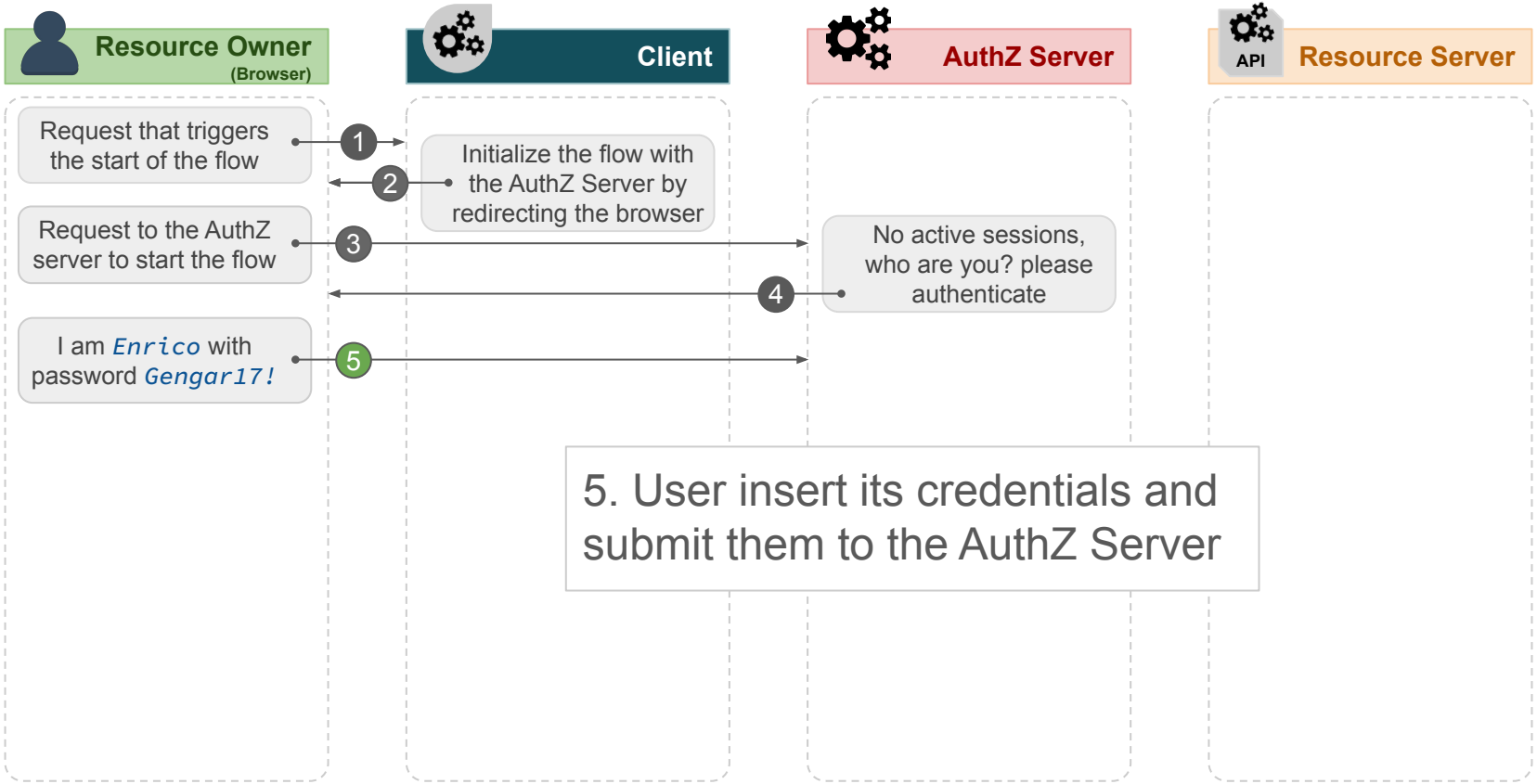
Grant Type / Flow name	Status
Implicit Flow	*Deprecated since OAuth 2.1*
Resource Owner Password Credentials Flow	*Discouraged since OAuth 2.1*
Authorization Code Flow	Commonly used
Client Credentials Flow	Commonly used
Refresh Token Flow	Commonly used
Device Code Flow	Commonly used

the **Authorization Code** flow

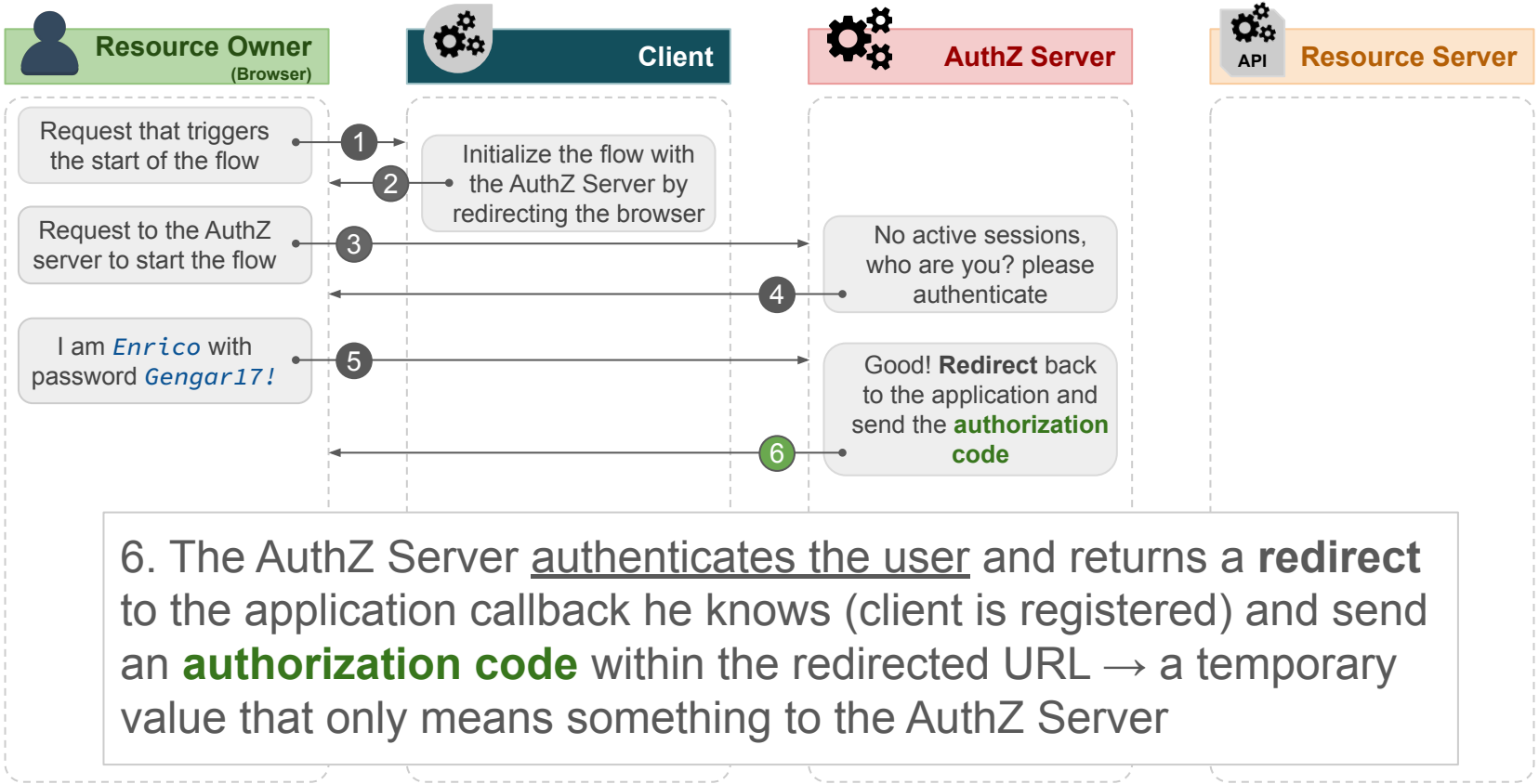


Authorization Code Grant
(Recommended for Most Apps)

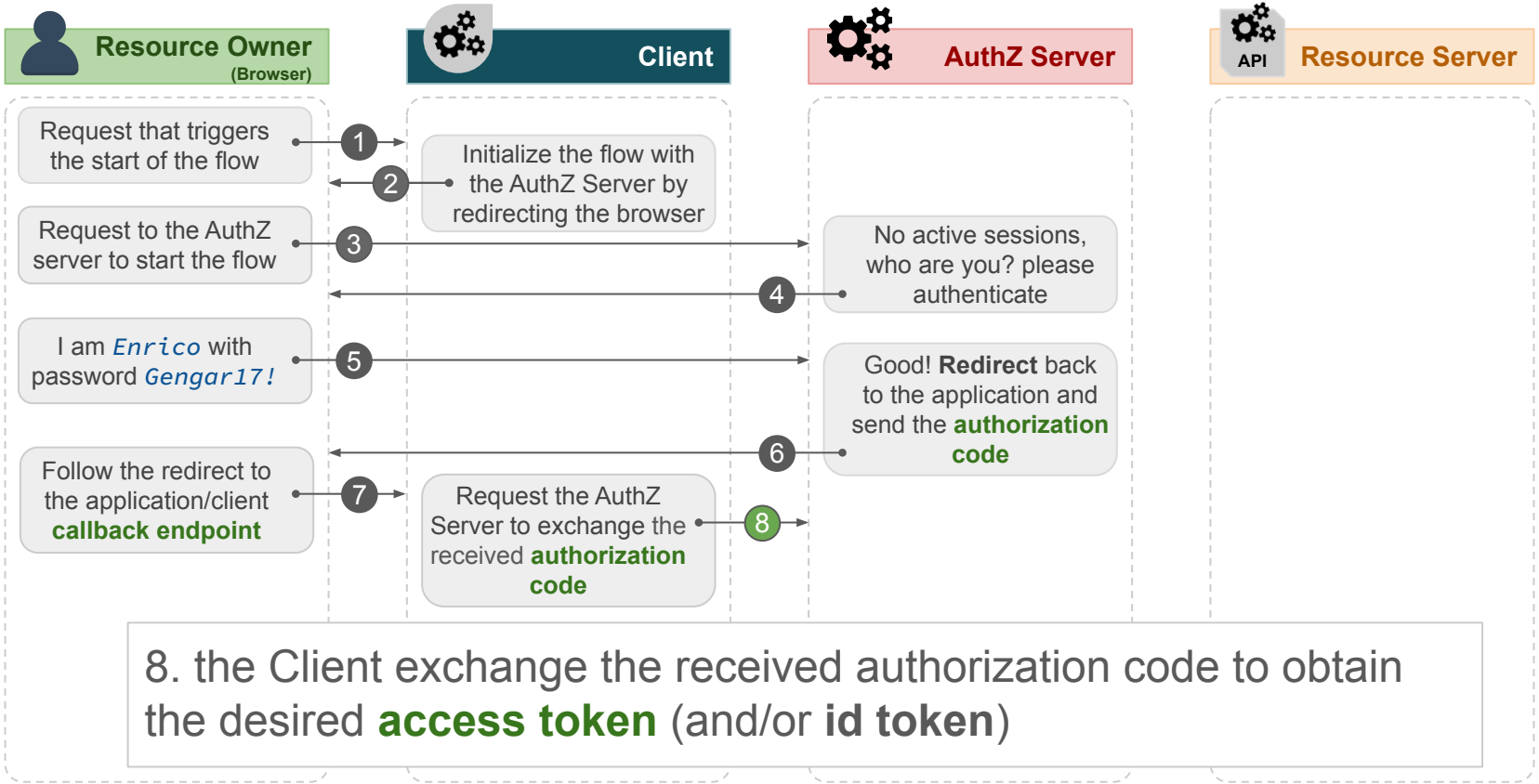

Resource Owner
(Browser)
Client
AuthZ Server
API
Resource Server
Request that triggers the start of the flow
1
Initialize the flow with the AuthZ Server by redirecting the browser
2
Request to the AuthZ server to start the flow
3
No active sessions, who are you? please authenticate
4
4. User is prompted to insert its credentials and to authorize the usage of its personal data
INFN CLOUD
Welcome to infn-cloud
Sign in with
INFN CCR - AAI
Local credentials
Not a member?
Apply for an account
Approval Required for Test Client
Client for IAM test client app
More information
Access to :
log in using your identity
basic profile information
email address
physical address
phone number
offline access
Remember this decision :
remember this decision until I revoke it
remember this decision for one hour
prompt me again next time
Authorizing will redirect to
https://iam-dev.cloud.cnaf.infn.it/iam-test-client/openid_connect_login
Authorize
Deny



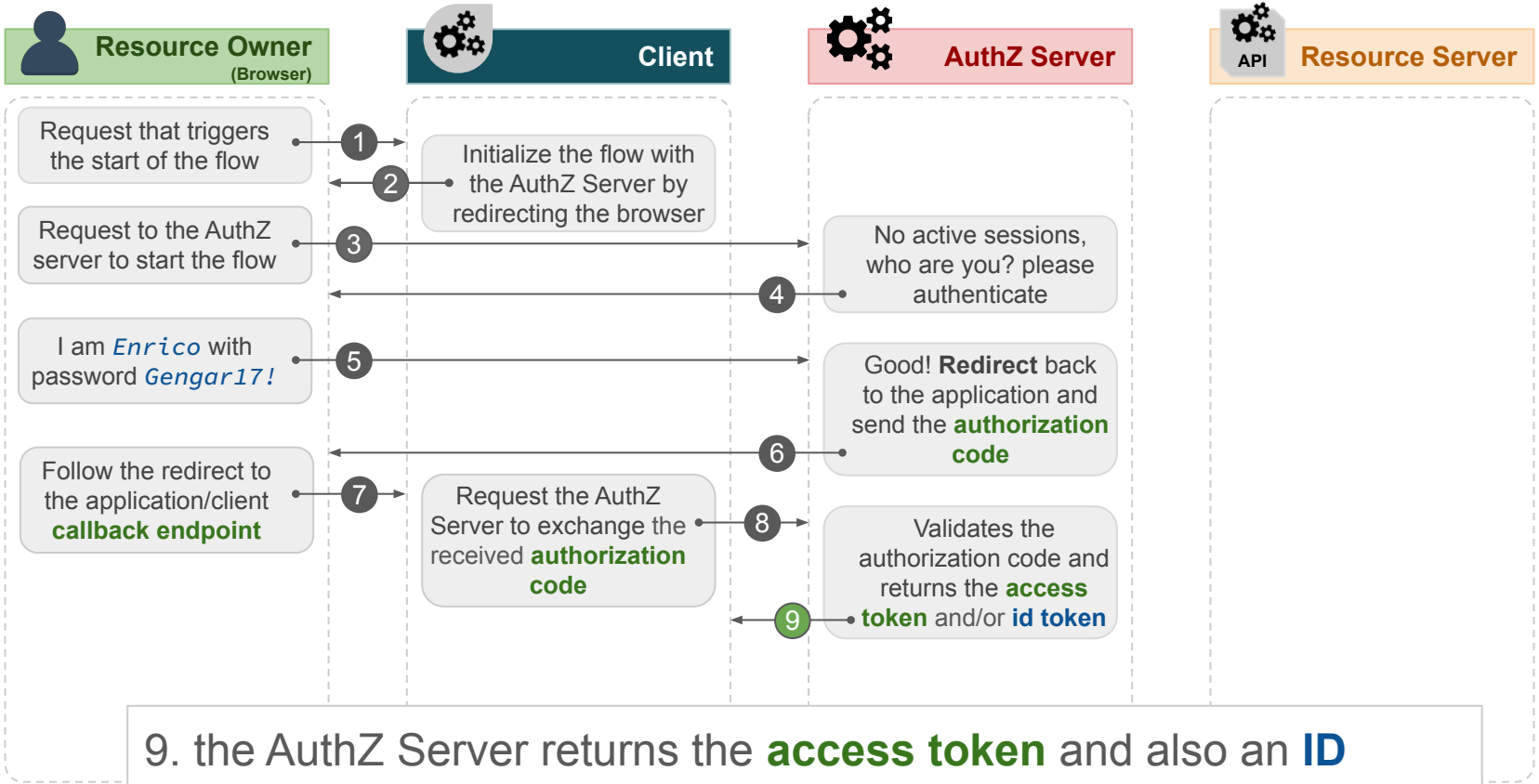
Authorization Code Grant
(Recommended for Most Apps)



Authorization Code Grant
(Recommended for Most Apps)

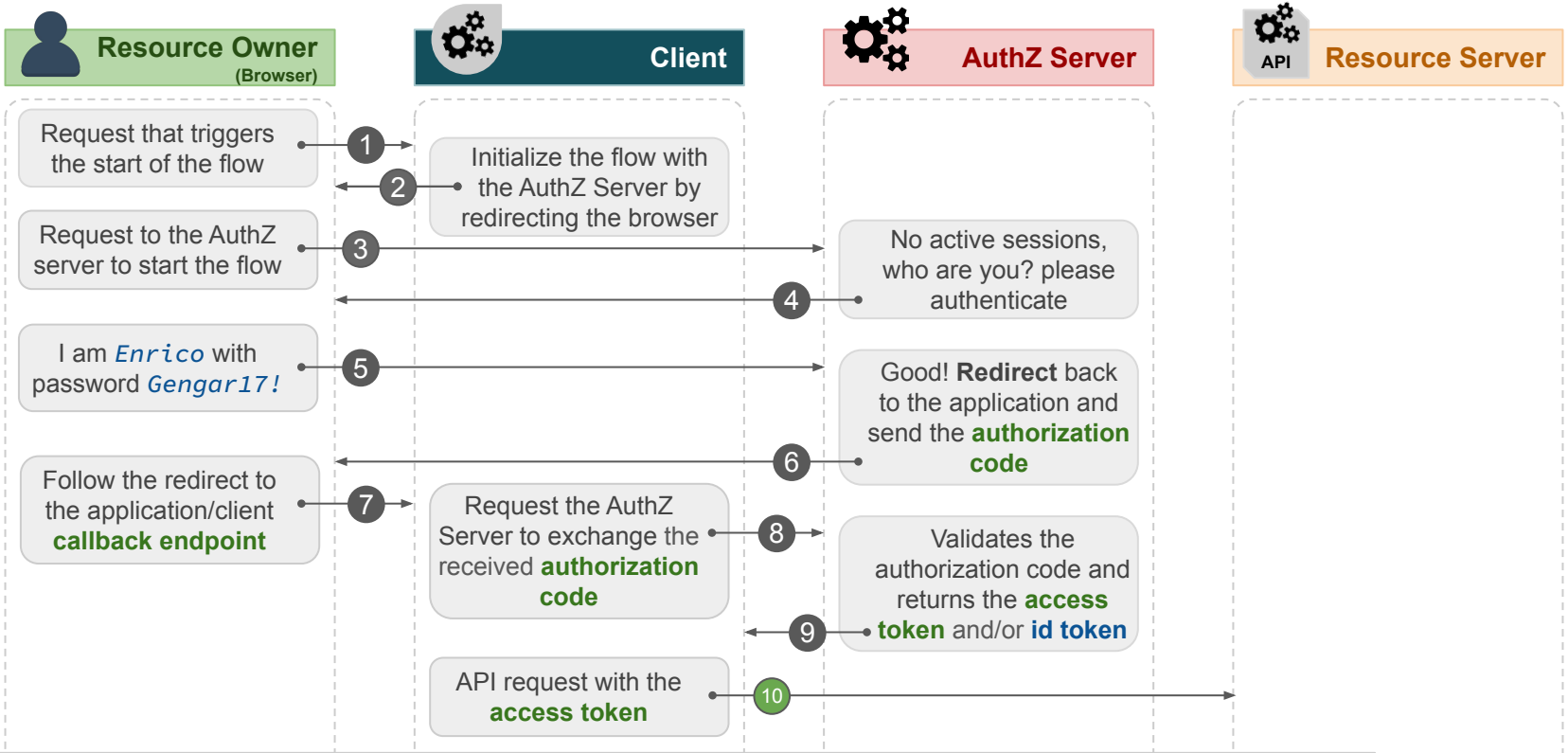


Authorization Code Grant
(Recommended for Most Apps)

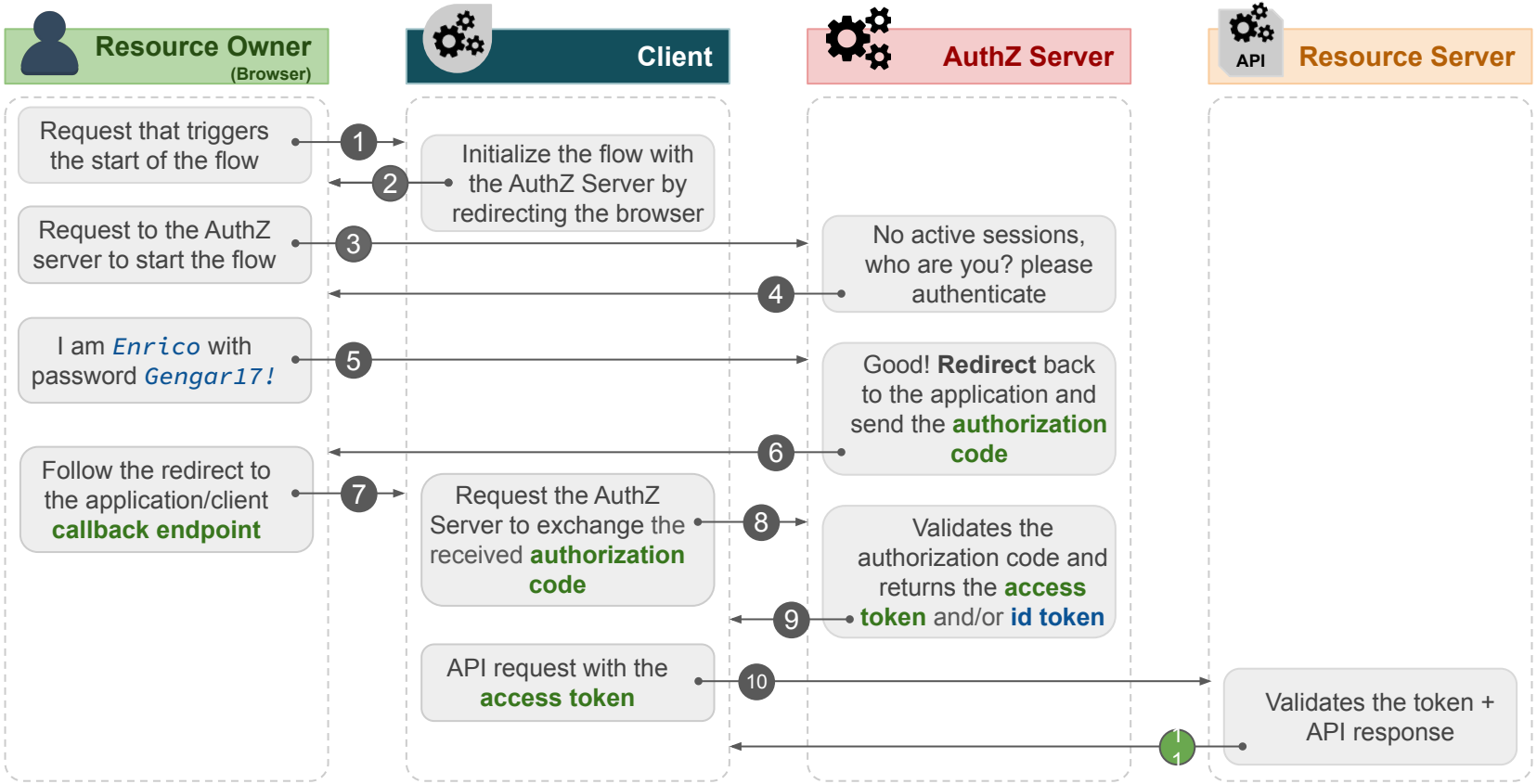


9. the AuthZ Server returns the **access token** and also an **ID Token** if it's an OpenID Provider

(Recommended for Most Apps)



10. the Client can finally **query the API endpoint** by using the received access token and grab the original requested data

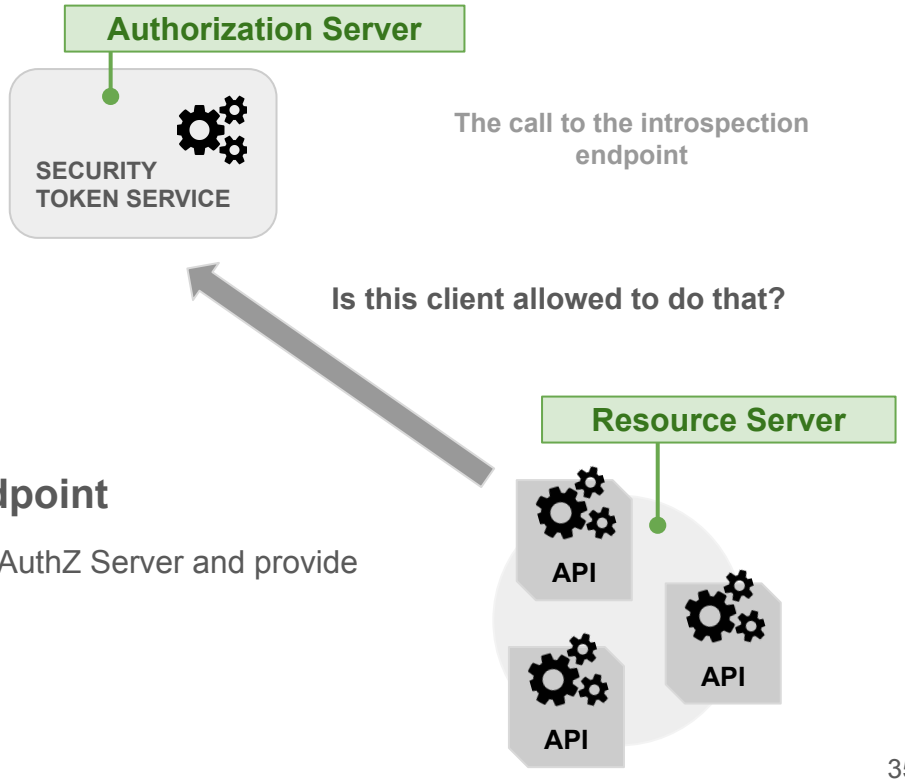


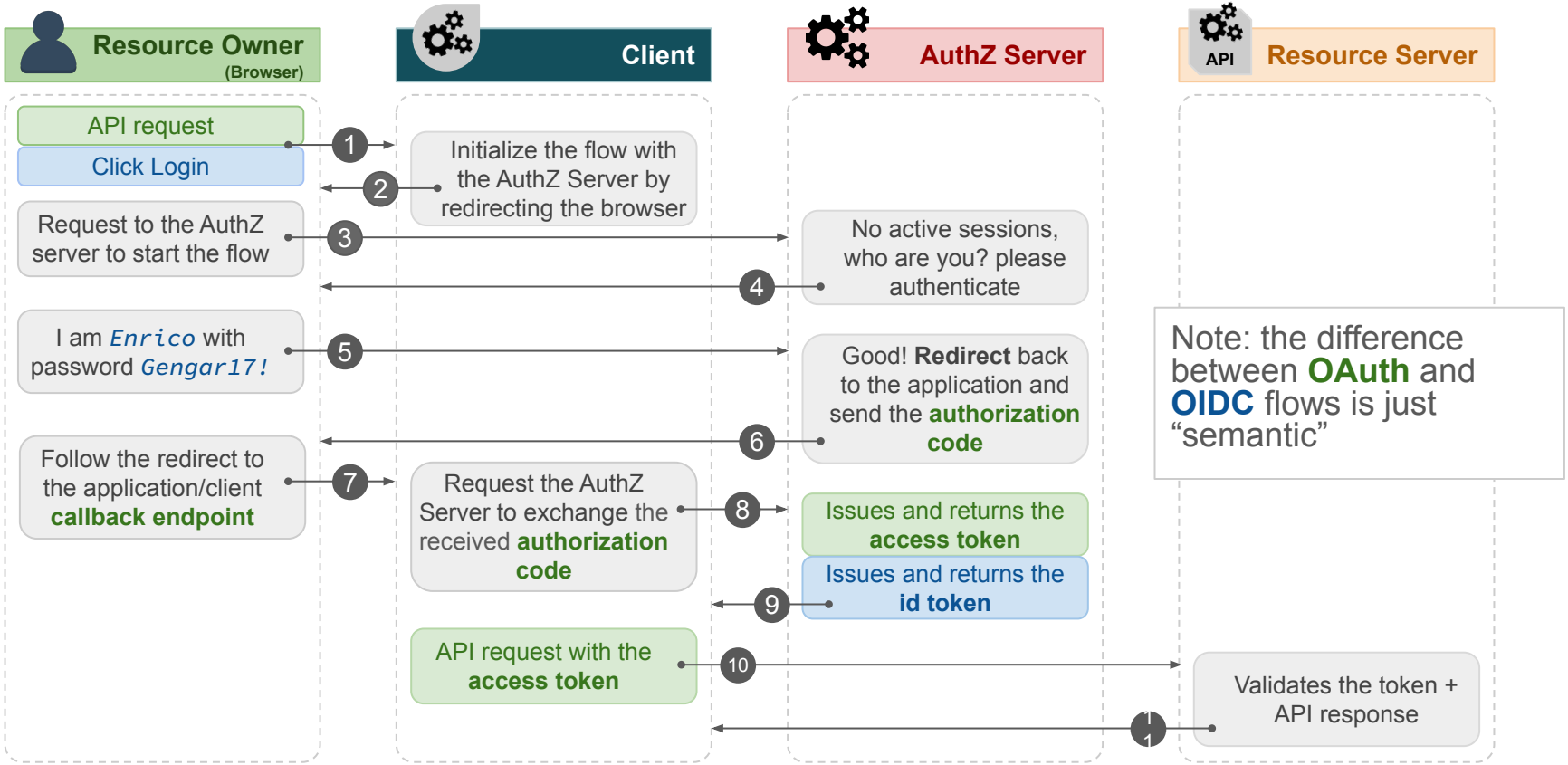
Authorization Code Grant
(Recommended for Most Apps)

OAuth2 token validation

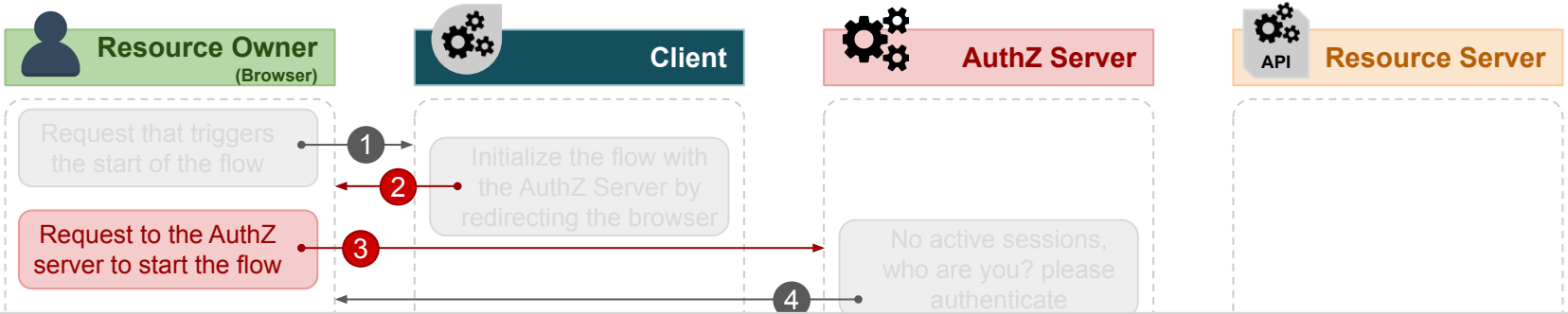
The Resource Server validates the access token:

- through **offline validation**
  - no credentials needed → no need to register the Resource Server on the Authorization Server
  - token is parsed and **the signature is verified** with **the public key exposed by the AuthZ Server**
- sending the token to the **introspection endpoint**
  - means the Resource Server is registered on the AuthZ Server and provide its credentials in order to do this
  - the answer is **valid: true/false**





Authorization Code Grant
(Recommended for Most Apps)



2 3 The authorization request (a redirect to the AuthZ Server)

https://iam.cloud.infn.it/authorize

?response_type=**code** → **indicates the authorization code flow**

&scope=**openid profile email** → **we want an ID token with email/profile info**

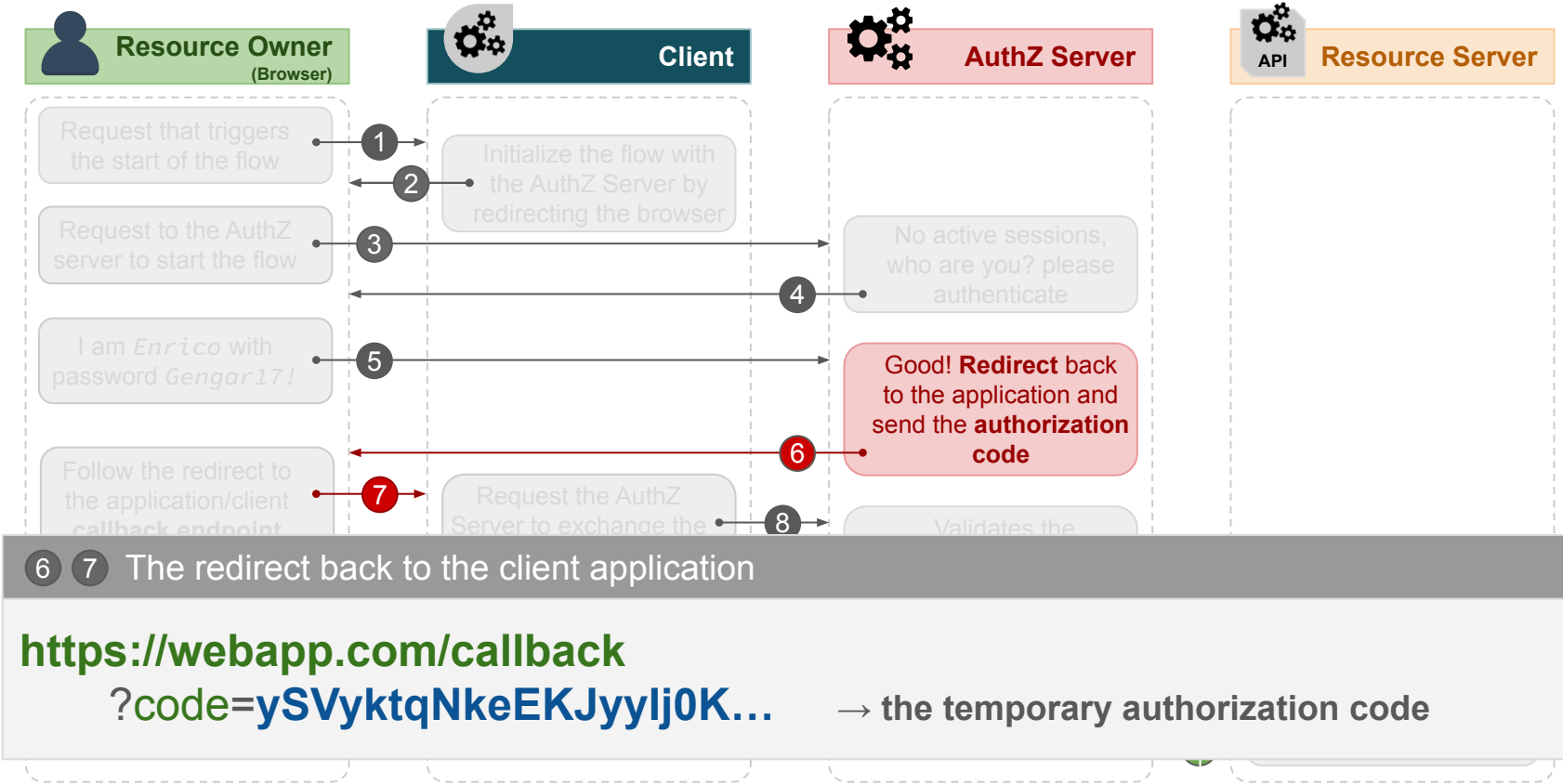
&client_id=**901C887F-EB2E-4957** → **the requestor (and registered) client ID**

&redirect_uri=**https://webapp.com/callback** → **the endpoint used on step 7**

&code_challenge=**F4554617…A353DC**

&code_challenge_method=**S256**

Authorization Code Grant
(Recommended for Most Apps)



6 7 The redirect back to the client application

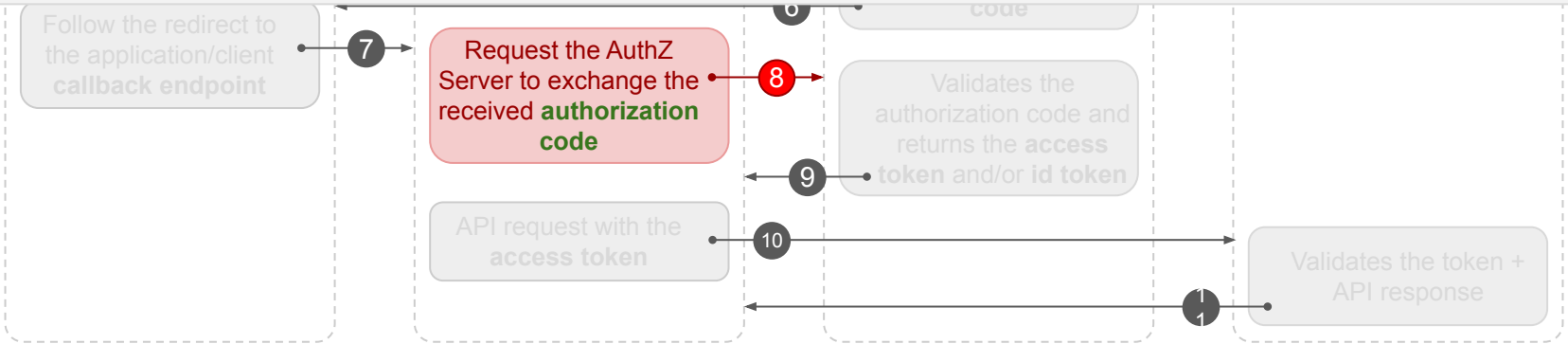
https://webapp.com/callback
?code=**ySVyktqNkeEKJyylj0K…** → the temporary authorization code

Authorization Code Grant
(Recommended for Most Apps)

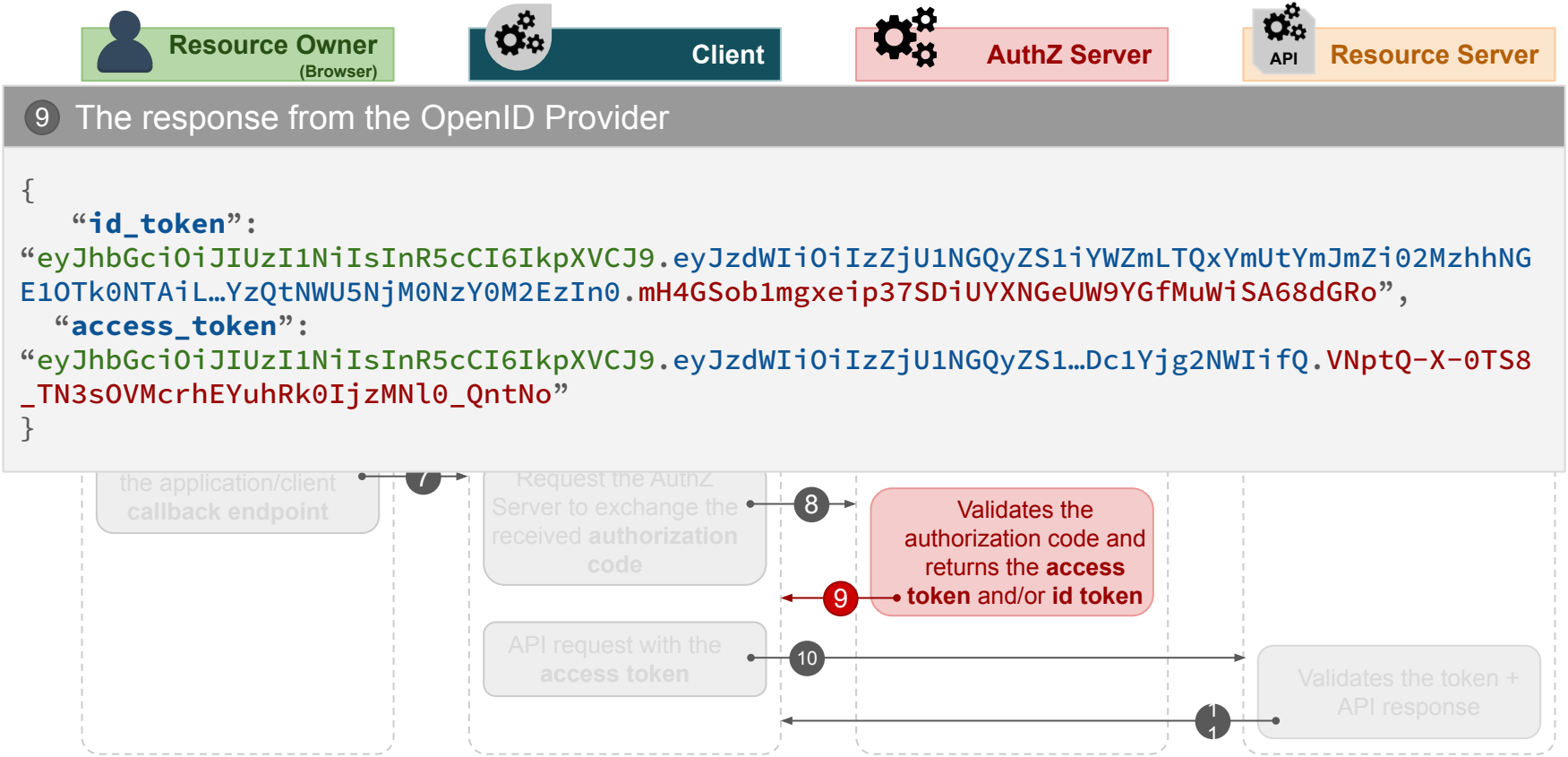
8 The request to exchange the authorization code

POST **https://iam.cloud.infn.it/oauth/token**

grant_type=**authorization_code** → **indicates the code exchange request**
&client_id=**901C887F-EB2E-4957** → **the confidential client exchanging the code**
&client_secret=**60DRv0g…0V0SWI** → **the client need to authenticate**
&redirect_uri=**https://webapp.com/callback** → **the redirect URI used before**
&code=**ySVyktqNkeEKJyyIj0K…** → **the authorization code received**
&code_verifier=**D0Hpp1yiK0iElVij…K8HBZB**



Authorization Code Grant
(Recommended for Most Apps)



Authorization Code Grant
(Recommended for Most Apps)

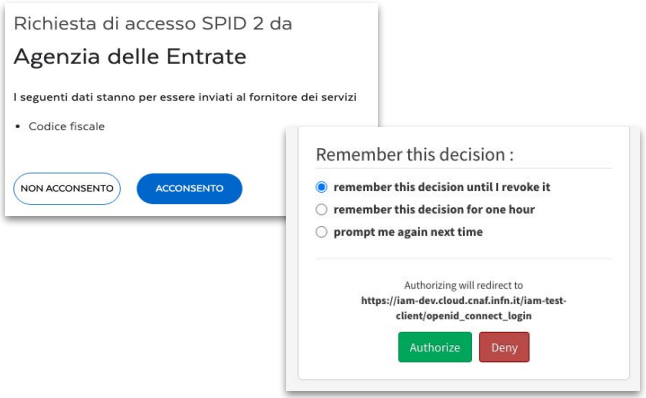
OAuth2 / OpenID Connect roles/terminology (3)

Scope: These are the **granular permissions** the Client wants, such as access to data or to perform actions.

https://webapp.com/callback

Redirect URI: The URL the Authorization Server will redirect the Resource Owner back to after granting permission to the Client. This is sometimes referred to as the "**Callback URL**."

OAuth2 / OpenID Connect roles/terminology (4)



Consent: The Authorization Server takes the Scopes the Client is requesting, and verifies with the Resource Owner whether or not they want to give the Client permission.

Client ID: Client identifier within the Authorization Server.

Client Secret: A secret password that only the Client and Authorization Server know. This allows them to securely share information privately behind the scenes.

OAuth2 / OpenID Connect roles/terminology (5)

Access Token:

- it's a string (→ **no format required**)
- clients use it to **make requests to the Resource Server**
- **it may be a bearer token** → those who hold the token can use it

ID Token:

- defined only within OIDC and it must be a **JWT**
- the OAuth Client should be the audience of the token
- **it contains information about the user such as name or email address**

Refresh Token (SPOILER):

- it's a string (→ **no format required**)
- clients use it to **get a new Access Token without the user's interaction**
- the new access token MUST have **a subset** of the original granted scopes
- **never** sent to Resource Servers

OAuth Bearer token usage

- RFC-6750
- It defines how to use tokens in HTTP requests to access protected resources on Resource Servers
- Any party in possession of a **bearer token** can use it to **get access** to the associated resources (without demonstrating possession of a cryptographic key)
- OAuth bearer token must be used in combination with **TLS over HTTP**
- Typically, tokens are sent in the **Authorization HTTP header**, as in the following example HTTP request

```
GET / HTTP/1.1
Host: apache.test.example
Authorization: Bearer eyJraWQiOiJy…rYI
User-Agent: curl/7.65.3
Accept: */*
```

The well-known endpoints

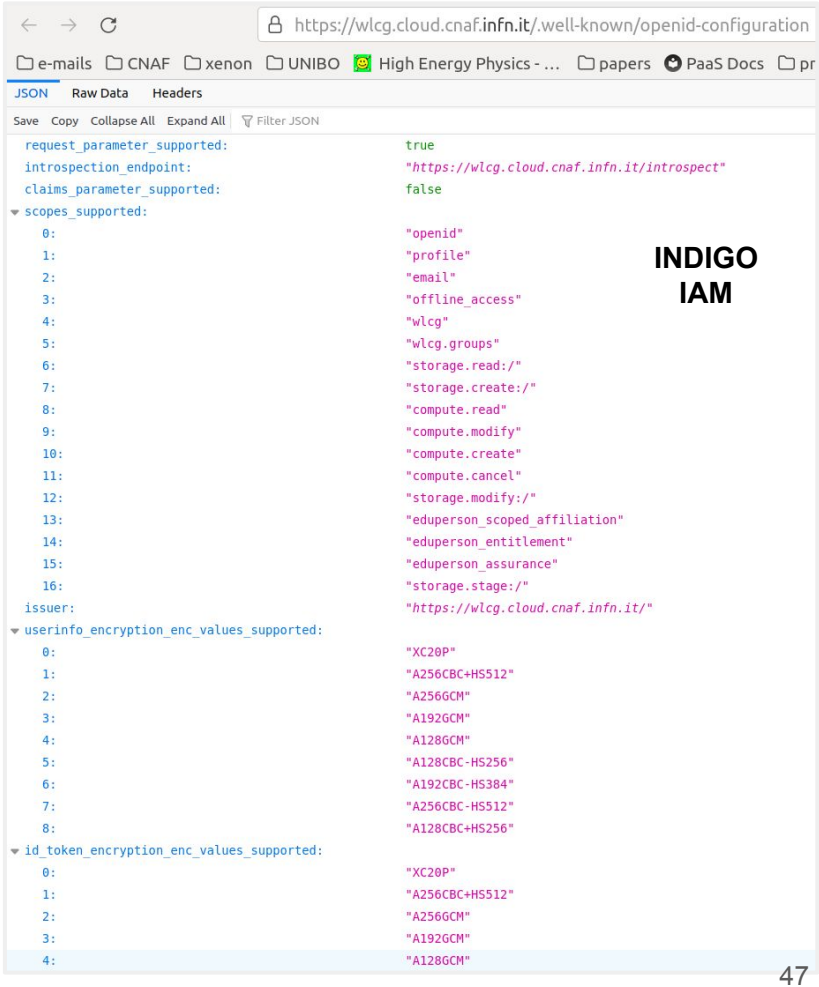
OAuth/OIDC provider metadata

- OAuth 2.0 / OpenID Connect specifications provide a standard way to expose their configuration
- Information is published at **a well-known endpoint**
  - `.well-known/openid-configuration` (if OIDC Provider)
  - `.well-known/oauth-authorization-server` (if AuthZ Server)
- Clients should use this information to know about
  - location of **key** used to **sign/encrypt tokens** → **used for token validation**
  - supported **grant types/authorization flows**
  - **endpoint** locations (authorize, token, dynamic client creation, etc.)
  - supported **scopes**
  - etc.

OAuth/OIDC provider metadata

Examples of metadata document:

- https://wlcg.cloud.cnaf.infn.it/.well-known/openid-configuration
- https://xfer.cr.cnaf.infn.it:8443/.well-known/openid-configuration
- https://xfer.cr.cnaf.infn.it:8443/.well-known/oauth-authorization-server
- https://accounts.google.com/.well-known/openid-configuration



```
{
  introspection_endpoint: "https://iam.cloud.infn.it/introspect",
  scopes_supported: [ "openid", "profile", "email", … , "offline_access" ],
  issuer: "https://iam.cloud.infn.it/",
  userinfo_encryption_enc_values_supported: [ "XC20P", "A256CBC+HS512", … , "A128CBC+HS256" ],
  id_token_encryption_enc_values_supported: [ "XC20P", "A256CBC+HS512", … , "A128CBC+HS256" ],
  authorization_endpoint: "https://iam.cloud.infn.it/authorize",
  device_authorization_endpoint: "https://iam.cloud.infn.it/devicecode",
  claims_supported: [
    "sub", "name", "preferred_username", "given_name", "family_name",
    "middle_name", "nickname", "profile", "picture", "zoneinfo", "locale",
    "updated_at", "email", "email_verified", "organisation_name", "groups" ],
  op_policy_uri: "https://iam.cloud.infn.it/about",
  token_endpoint_auth_methods_supported: [
    "client_secret_basic", "client_secret_post", "client_secret_jwt", "private_key_jwt", "none" ],
  token_endpoint: "https://iam.cloud.infn.it/token",
  response_types_supported: [ "code", "token" ],
  grant_types_supported: [ "authorization_code", "implicit", "refresh_token", "client_credentials",
    "password", "urn:ietf:params:oauth:grant-type:token-exchange", "urn:ietf:params:oauth:grant-type:device_code" ],
  revocation_endpoint: "https://iam.cloud.infn.it/revoke",
  userinfo_endpoint: "https://iam.cloud.infn.it/userinfo",
  op_tos_uri: "https://iam.cloud.infn.it/about",
  token_endpoint_auth_signing_alg_values_supported: [ "HS256", "HS384", … , "PS512" ],
  require_request_uri_registration: false,
  code_challenge_methods_supported: [ "plain", "S256" ],
  id_token_encryption_alg_values_supported: [ "RSA-OAEP-512", "RSA-OAEP", … , "RSA-OAEP-384" ],
  jwks_uri: "https://iam.cloud.infn.it/jwk",
  subject_types_supported: [ "public", "pairwise" ],
  id_token_signing_alg_values_supported: [ "HS256", "HS384", … , "none" ],
  registration_endpoint: "https://iam.cloud.infn.it/iam/api/client-registration",
}
```

OIDC userinfo endpoint

The UserInfo endpoint is an OAuth 2.0 protected resource where client applications can retrieve a JSON object that contains claims about the logged in end-user. The sub member represents the subject (end-user) identifier.

The content of this JSON object can overlap with the content of an ID token.

Clients **must present a valid access token** to retrieve the claims.

The UserInfo endpoint is described in the OpenID Connect Core 1.0 specification.

```
{
  "sub"         : "83692",
  "name"        : "Alice Adams",
  "given_name"  : "Alice",
  "family_name" : "Adams",
  "email"       : "alice@example.com",
  "picture"     : "https://example.com/83692/photo.jpg"
}
```

JSON Web Tokens (JWT)

JSON **W**eb **T**okens (JWTs)

- JSON Web Token is a compact, self-contained way of securely transmitting information between parties in a **JSON object**
- A JWT is represented as a sequence of **URL-safe parts** separated by period (“.”) characters. Each part contains a **base64url-encoded value**.
- The number of parts in the JWT is dependent upon the representation of the resulting JSON Web Signature (JWS) using the JWS Compact Serialization or JSON Web Encryption (JWE) using the JWE Compact Serialization
    - Typically: header, payload, and signature
- The **payload** of the JWT is encoded in **token claims**
- JWTs are typically **signed** and, if confidentiality is a requirement, can be **encrypted**
- Main specification: RFC 7519

JWT: Header.Payload.Signature

Example of encoded token

eyJraWQiOiJyc2ExIiwiYWxnIjoiUlMyNTYifQ.eyJ3bGNnLnZlciI6IjEuMCIsInN1YiI6IjBmZDc2YjNjLWMzZjEtNDI4MC1iZTNjLTVlYmVhZDgxYzZkNiIsImF1ZCI6Imh0dHBzOlwvXC93bGNnLmNlcm4uY2hcL2p3dFwvdjFcL2FueSIsIm5iZiI6MTY2OTEyNzI3Nywic2NvcGUiOiJzdG9yYWdlLnJlYWQ6XC8iLCJpc3MiOiJodHRwczpcL1wvd2xjZy5jbG91ZC5jbmFmLmluZm4uaXRcLyIsImV4cCI6MTY2OTEzMDg3NywiaWF0IjoxNjY5MTI3Mjc3LCJqdGkiOiI5ZDE0NGRhMC1hMTQ5LTQwZTItYWM3NS01MjM0YzFjOTcyODIiLCJjbGllbnRfaWQiOiJlYjllMWNjMi1mNWUxLTRhNGItYjk2Ny1iY2NlYTI2NmYwOWIifQ.YbsCossZBloBxJBgk9D-IdVuAzm67rl_MVVdp8j4bXicLgPCM-6Wdze2VMzR6Nw0KMCBXhs59e5glgq0Fr5kagrpPjuua2sHX5ul84SNvlgoKMwSn_NIDXSO9flaDIIuelrSgT1qOTSiMV5M_U4VpWjOimpYm9fxmLSSIZT59MU

JWT: Header.Payload.Signature

Example of decoded token

Header

```
$ echo $AT | cut -d. -f1 |
base64 -d 2>/dev/null | jq

{
  "kid": "rsa1",
  "alg": "RS256"
}
```

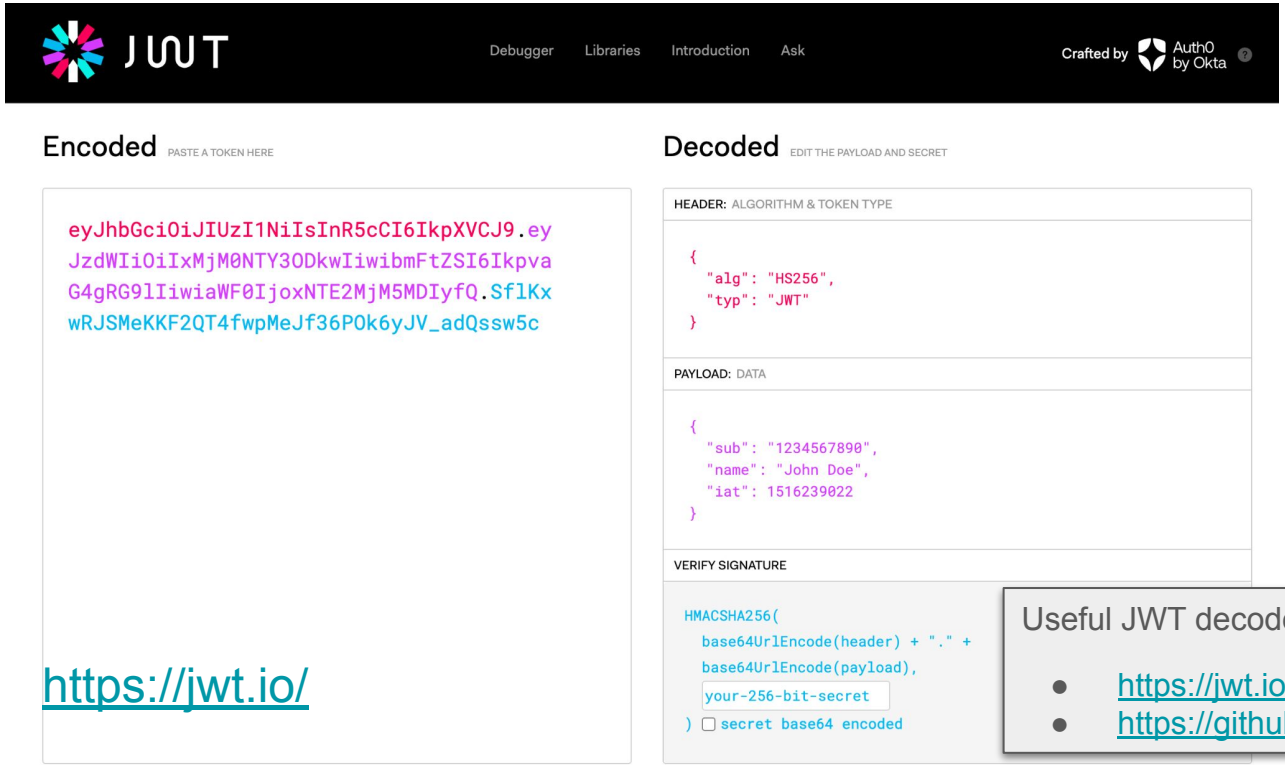
Payload

```
$ echo $AT | cut -d. -f2 | base64 -d
2>/dev/null | jq

{
  "wlcg.ver": "1.0",
  "sub":
"0fd76b3c-c3f1-4280-be3c-5ebead81c6d6",
  "aud": "https://wlcg.cern.ch/jwt/v1/any",
  "nbf": 1669127273,
  "scope": "storage.read:/",
  "iss": "https://wlcg.cloud.cnaf.infn.it/",
  "exp": 1669130873,
  "iat": 1669127273,
  "jti":
"2222be79-e218-442b-9389-c741c5b95da2",
  "client_id":
  "eb9e1cc2-f5e1-4a4b-b967-bccea266f09b"
}
```

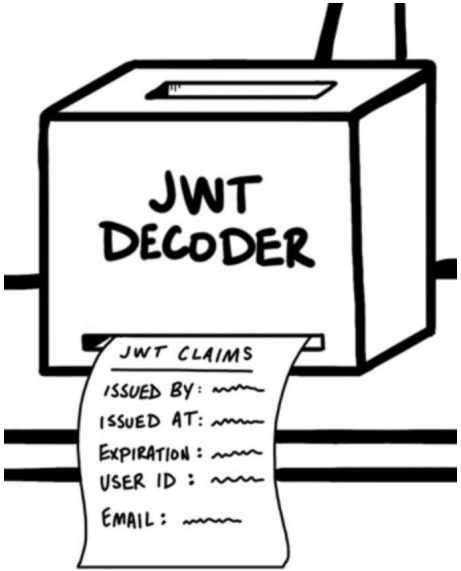
Signature

```
$ echo $AT | cut -d. -f3

Zcamp7C40T4oygiO9_ua6oASnE
TYvkZhr8x_OredqLQagryptTwl
iDJRcCA2L8Uff_Tyh8KxKJsc1e
k86pGEZnkckFcfKscNJQyg8qKt
4plTDpxUkMV0ficF--IFOK3ACl
u18kWSG1pc85IGl8r64qF5e46o
fHjblGDnQAz06bc
```

JWT utilities



https://jwt.io/



Useful JWT decoders

- https://jwt.io/
- https://github.com/troyharvey/jwt-cli

JWT claim names

Typical registered claim names (i.e. a set of basic claims defined by the JWT standard)

- “**iss**” (Issuer): the principal (Authorization Server/OpenID Provider) that issued the JWT (e.g., iam.cloud.infn.it)
- “**sub**” (Subject): the principal that is the subject of the JWT (e.g., a unique ID linked to an IAM account)
- “**aud**” (Audience): identifies the recipients that the JWT is intended for (e.g., hostname of a RUCIO instance)
- “**exp**” (Expiration time): identifies the expiration time after which the JWT MUST NOT be accepted by resources
- “**nbf**” (Not before): identifies the time before which the JWT MUST NOT be accepted by resources
- “**iat**” (Issued at): identifies the time at which the JWT was issued
- “**jti**” (JWT ID): provides a unique identifier for the JWT

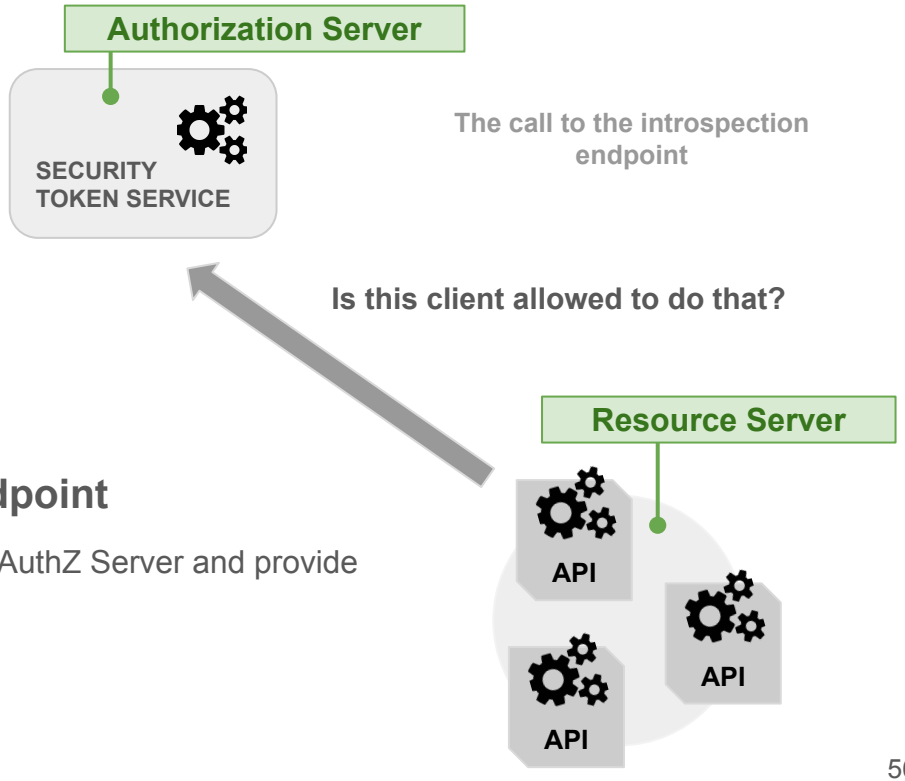
Additional INDIGO IAM claims (SPOILER)

- “**client_id**”: ID of the client which requests the token
- “**scope**”: list of token capabilities
- “**groups**”: list of groups the user is member of

OAuth2 token validation

The Resource Server validates the access token:

- through **offline validation**
  - no credentials needed → no need to register the Resource Server on the Authorization Server
  - token is parsed and **the signature is verified** with **the public key exposed by the AuthZ Server**
- sending the token to the **introspection endpoint**
  - means the Resource Server is registered on the AuthZ Server and provide its credentials in order to do this
  - the answer is **valid: true/false**



JWT offline validation

Section 4 of RFC 9068

The **Offline Validation** of a JWT means:

- check that the **current time** is **before** the time represented by the “**exp**” claim
  - delays of few minutes are allowed to account for clock skew
- the authorization server **issuer** identifier **MUST exactly match** the value of the "**iss**" claim
- the **signature** MUST be checked using the algorithm specified in the JWT "**alg**" Header Parameter
  - the well-known endpoint of the AuthZ Server shares its public/symmetric key through the ***jwks_uri*** field
- if validation is performed by the Resource Server, the **“aud”** claim must contain a resource indicator value corresponding to the **Resource Server itself**

Examples of scopes

Standard commonly used OAuth/OIDC scopes

- **openid** signal that the Client wants to receive authentication information about the user → the **ID Token**
- **profile** used to request **profile information** (name, address, *etc*)
- **email** used to request access to the **user's email**
- **offline_access** used to **request refresh tokens**

Proof Key for Code Exchange (PKCE) and Authorization Code Flow

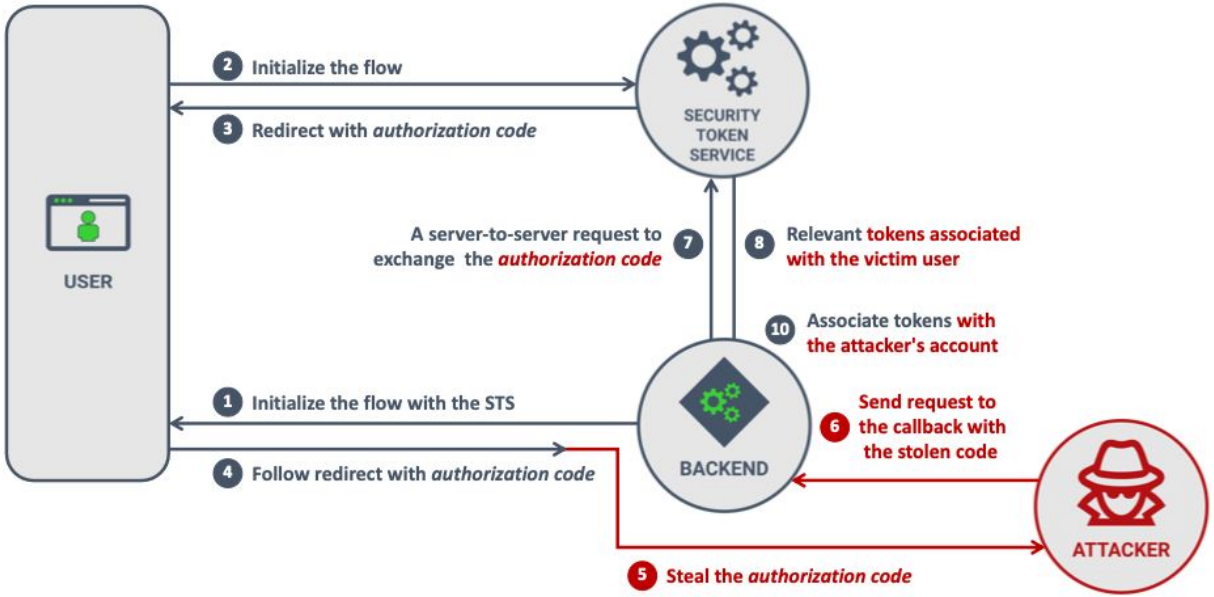
The Authorization Code Flow

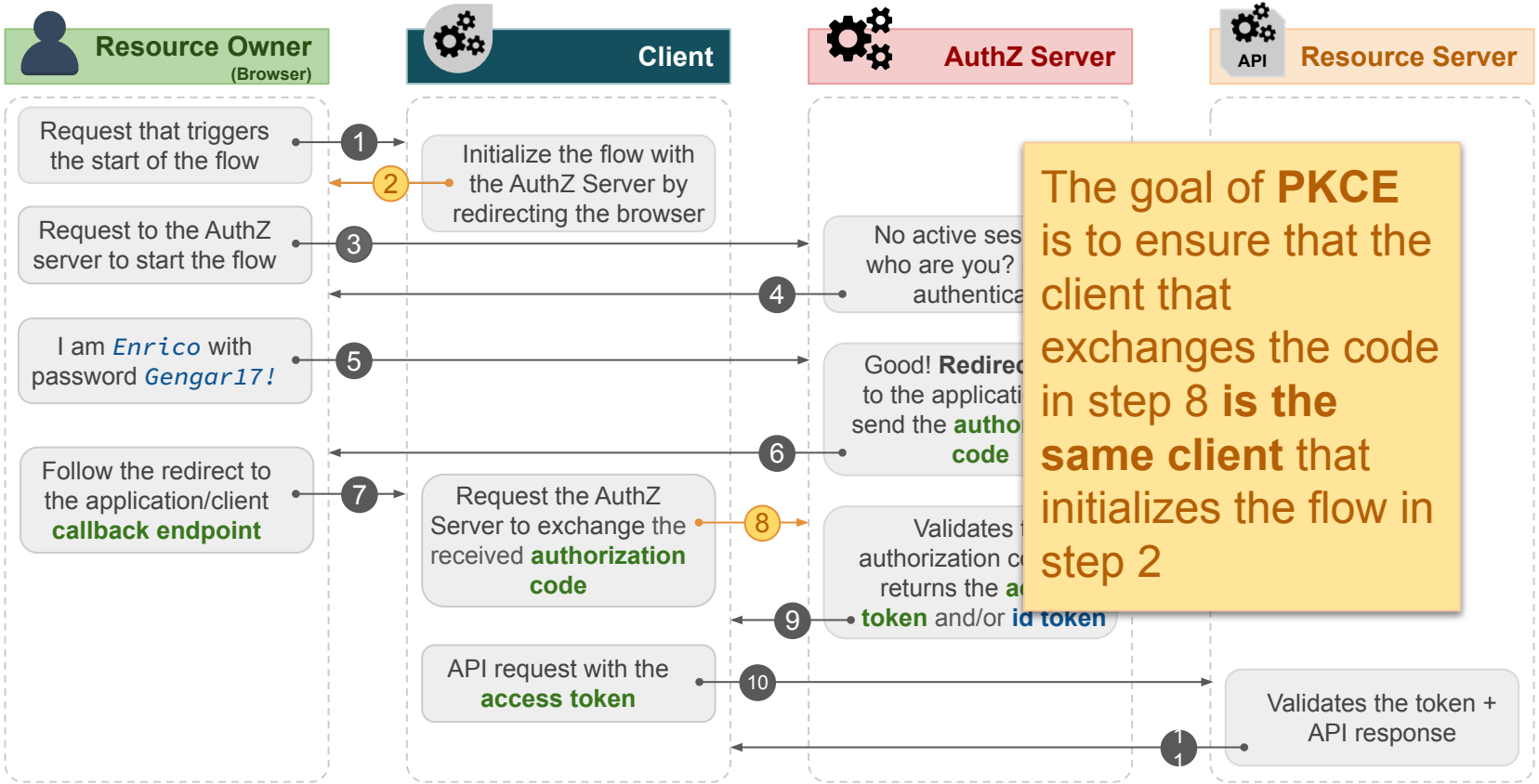
- The Authorization Code flow supports both OAuth and OIDC scenarios
  - The ***openid*** scope augments the OAuth Authorization Code flow with OIDC features
- The client application is known as a **confidential client**
  - Confidential clients run in a restricted environment (e.g., a server environment)
  - Confidential clients have access to a secret, allowing them to authenticate to the AS
- The authorization code is protected against abuse
  - A confidential client needs to authenticate to exchange an authorization code
  - Authorization codes should be **short-lived** and should only be **valid for one-time use**

An Authorization Code INJECTION ATTACK

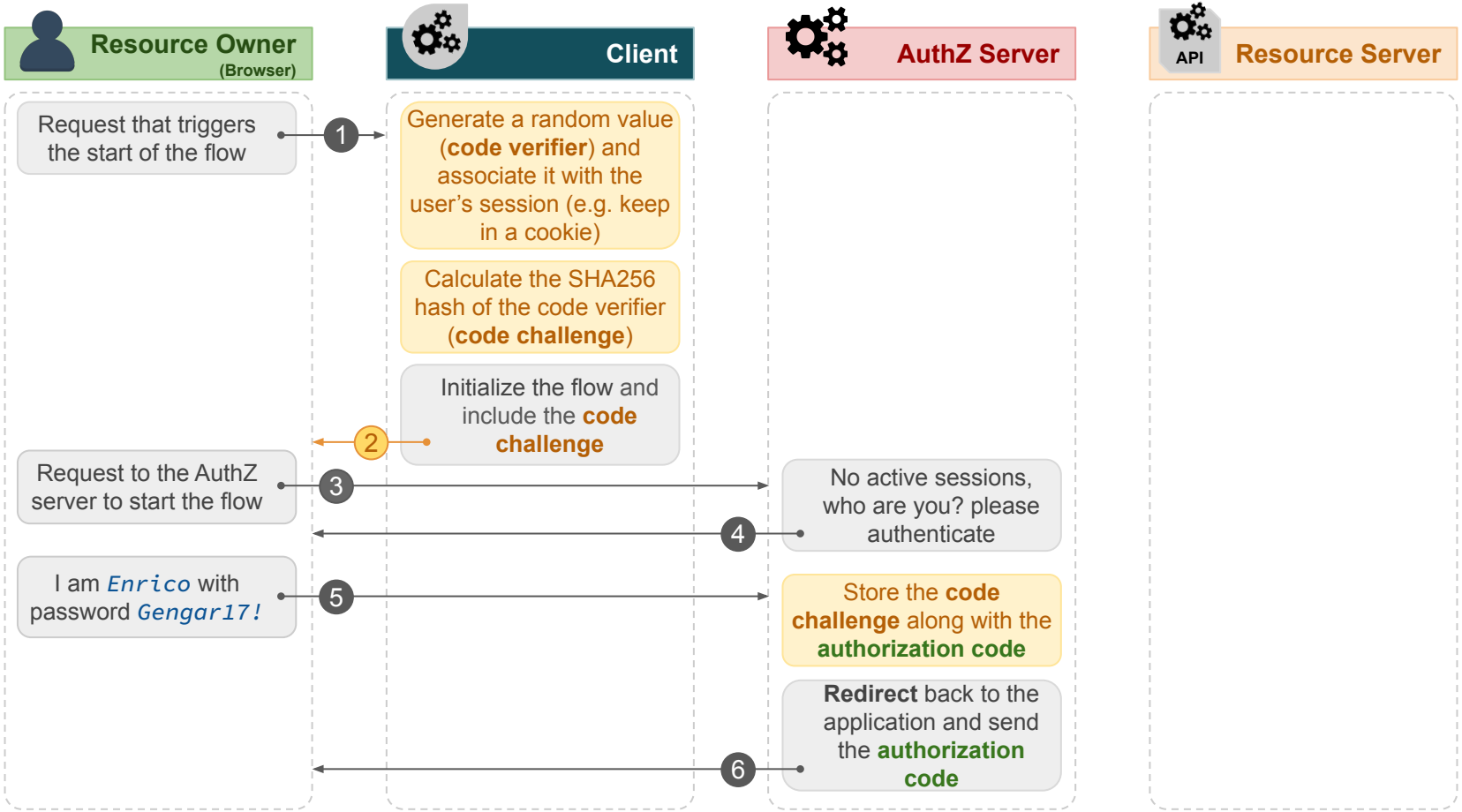
The Authorization Code flow relies on the insecure front channel to relay the code and this could have consequences →

Proof Key for Code Exchange (**PKCE**) helps protect the integrity of the Authorization Code flow

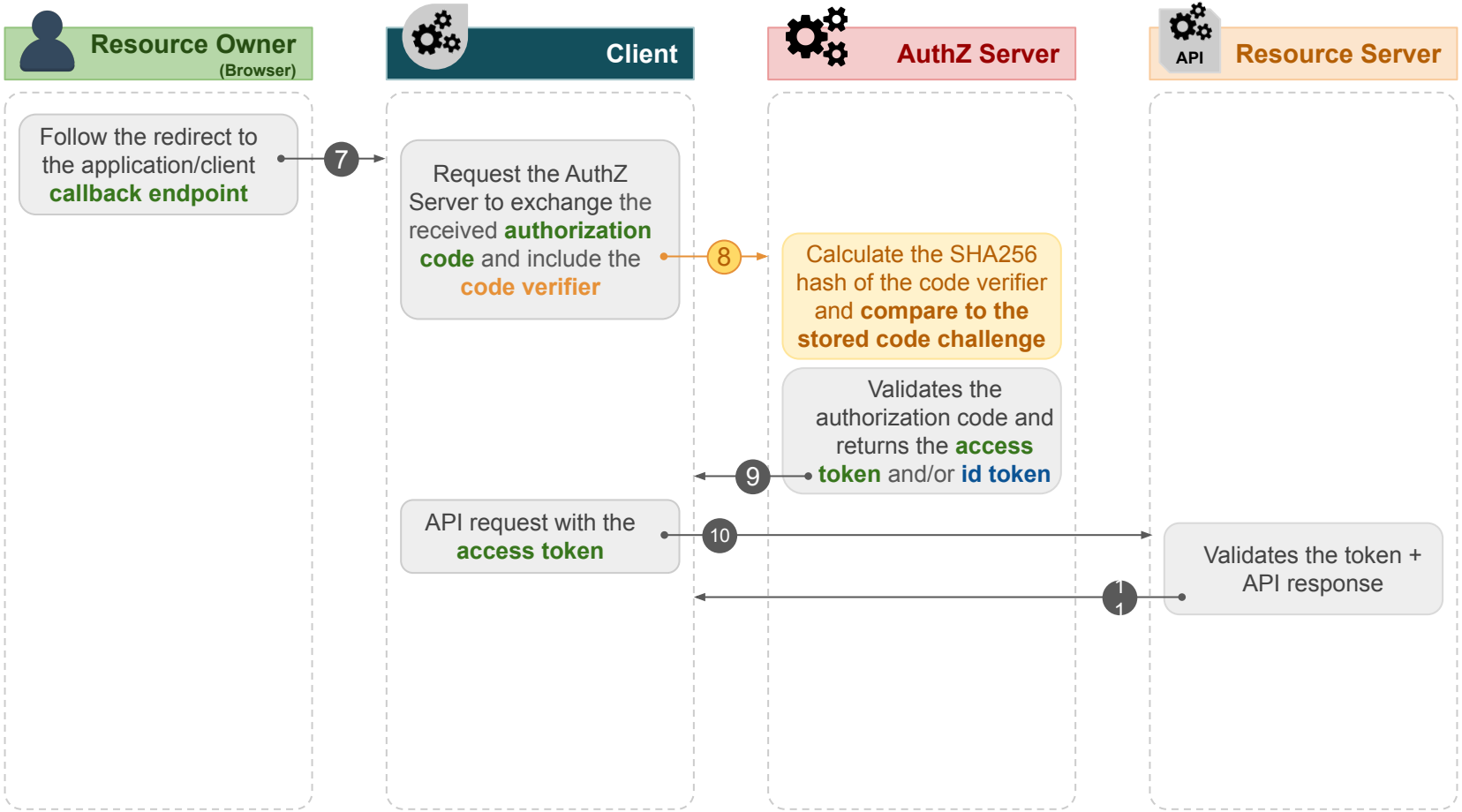




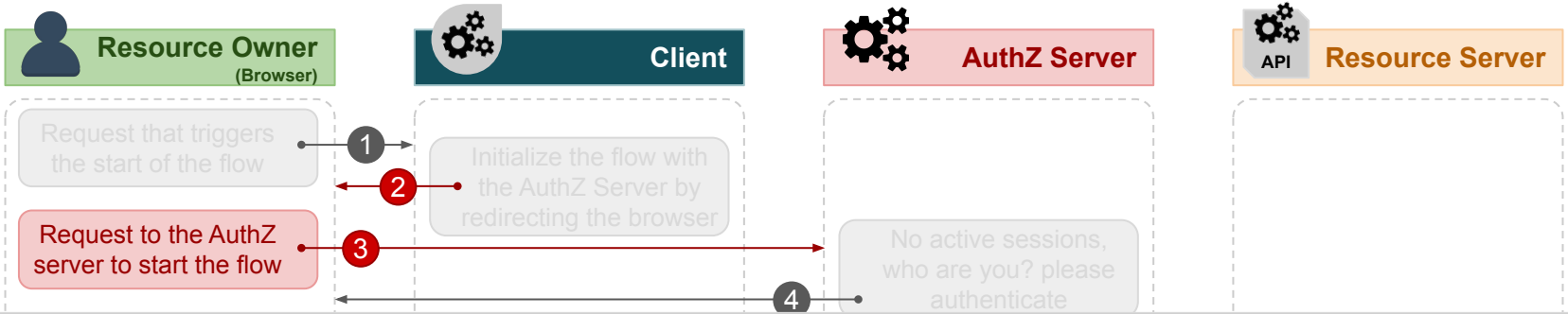
Authorization Code Grant
(Recommended for Most Apps)



Authorization Code Grant with PKCE



Authorization Code Grant with PKCE



2 3 The authorization request (a redirect to the AuthZ Server)

https://iam.cloud.infn.it/authorize

?response_type=**code**

&scope=**openid profile email**

&client_id=**901C887F-EB2E-4957**

&redirect_uri=**https://webapp.com/callback**

&code_challenge=**F455…A353DC** → **the code challenge (hash of code verifier)**

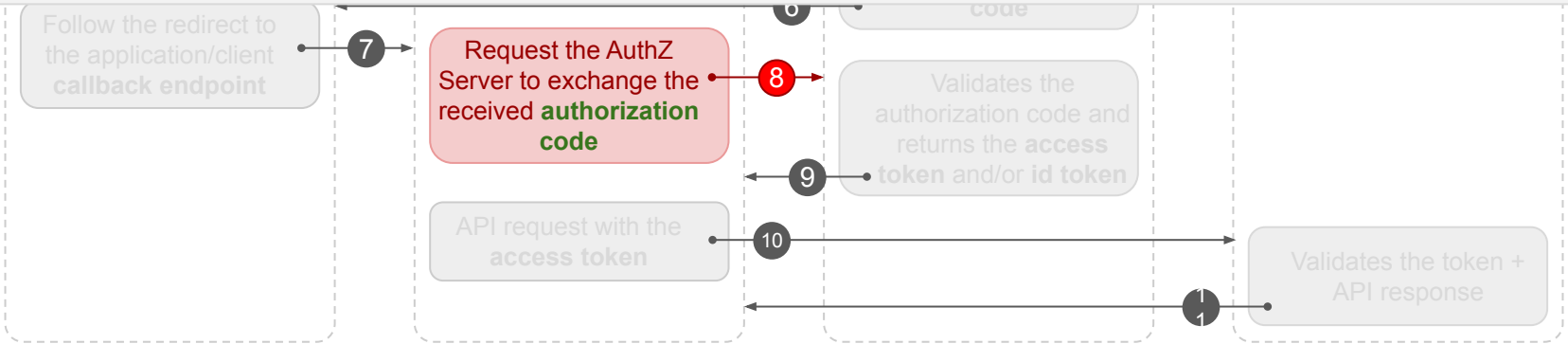
&code_challenge_method=**S256** → **the hash function**

Authorization Code Grant
(Recommended for Most Apps)

8 The request to exchange the authorization code

POST **https://iam.cloud.infn.it/oauth/token**

grant_type=**authorization_code**
&client_id=**901C887F-EB2E-4957**
&client_secret=**60DRv0g…0V0SWI**
&redirect_uri=**https://webapp.com/callback**
&code=**ySVyktqNkeEKJyyIj0K…**
&code_verifier=**D0Hpp1yiK0iElVij…K8HBZB** → **the code verifier from step 2**



Authorization Code Grant
(Recommended for Most Apps)

Proof Key for Code Exchange (PKCE)

- PKCE consists of a code verifier and a code challenge
  - The **code verifier** is a **cryptographically secure random string**
    - Between 43 and 128 characters of this character set: [A-Z] [a-z] [0-9] - . _ ~
  - The **code challenge** is a **base64 urlencoded SHA256 hash** of the **code verifier**
    - The hash function uniquely connects the code challenge to the code verifier
    - The code verifier cannot be derived from the code challenge
- PKCE ensures that the same client intializes and finalizes the flow
  - PKCE was originally intended to secure flows of public clients (no client authentication)
  - Today, PKCE **is a recommended best practice** to guarantee flow integrity
- **PKCE replaces the OAuth *state* parameter or OIDC *nonce* for security**

Token based Authorization

Token-based Authorization

We understood that:

- to **access resources**/services → a Client application needs an **Access Token**
- to **identify a user** → a Client applications needs an **ID Token**
- the tokens are obtained from an Authorization Server|OIDC Provider using standard OAuth|OIDC flows

Authorization is performed at the Resource Server level, leveraging info extracted from the token:

- **Identity attributes**: e.g., **groups** included into ID token (and optionally also into access token → it can contain whatever the provider wants)
- **Scopes**: capabilities linked to access tokens at token creation time (each provider can define its own scopes)

Identity-based vs Scope-based Authorization

Identity-based authorization

- the **ID token** brings information about attribute entitlement (*e.g.*, group/role membership)
- the service maps these attributes to a local authorization policy

Scope-based authorization

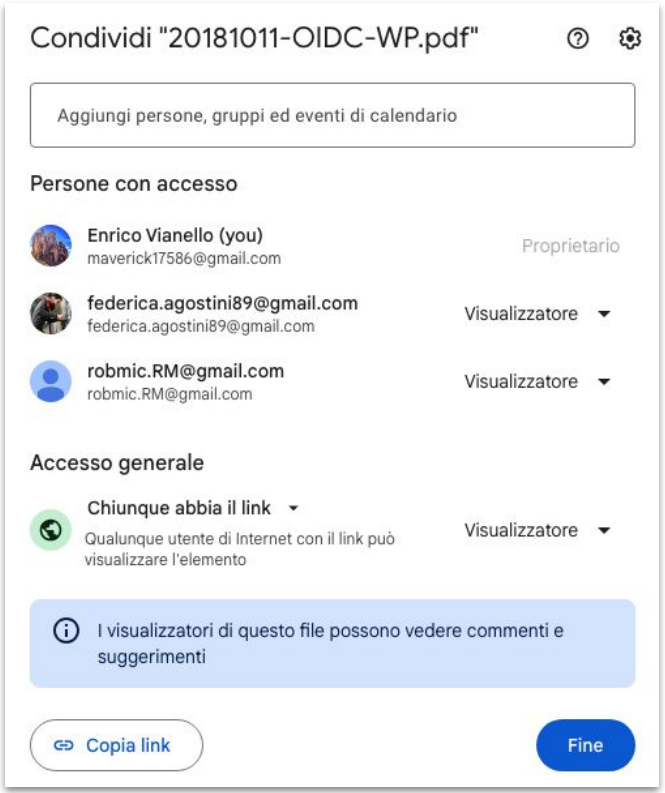
- the **access token** brings information about which actions should be authorized at a service
- the service needs to understand these capabilities **and honor them**
- the authorization policy is managed at the VO level (*i.e.*, IAM)

Identity-based vs Scope-based Authorization

The two models can coexist, even in the context of the same application

identity based AuthZ →

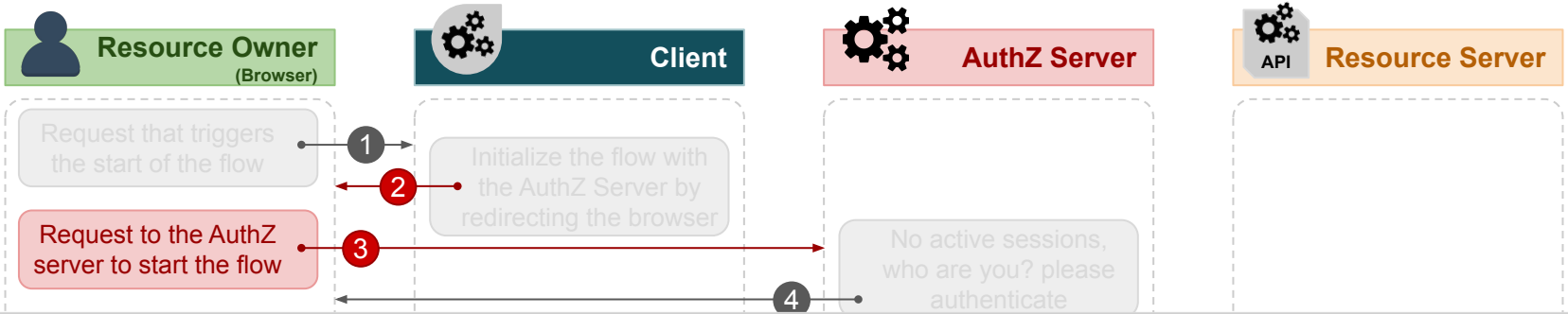
scope based AuthZ →



Long-Term access with Refresh Tokens

Refresh Token flow

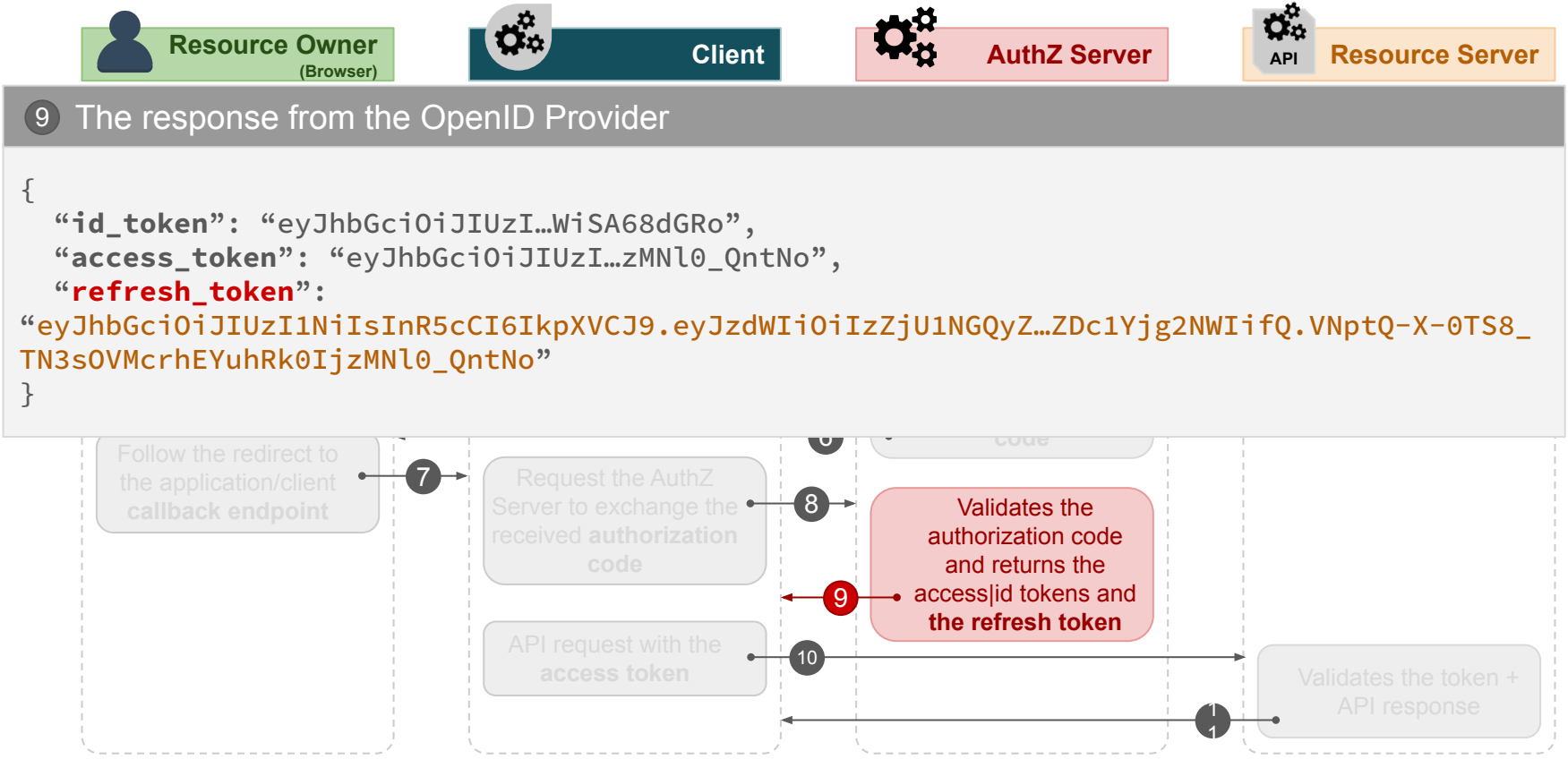
- RFC-6749 Section 1.5
- The application acts **on behalf of a user** and get a new access token without any user interaction → e.g. to refresh an Access Token that is about to expire
- It starts with an authenticated POST to the Authorization Server token endpoint
  - **Client must authenticate**
  - A valid **Refresh Token** must be provided
  - A new Access Token and possibly an updated Refresh Token are returned
- A Refresh Token request can be performed in order **to change the audience claim** in place of using the token exchange flow (we'll see it later)
- The flow authorization grant is a **refresh token**, obtained after an authorization flow (e.g. code, see next slide)



2 3 The authorization request (a redirect to the AuthZ Server) if we want a refresh token too

https://iam.cloud.infn.it/authorize
?response_type=**code**
&scope=**openid profile email offline_access** → **ask for a refresh token too**
&client_id=**901C887F-EB2E-4957**
&redirect_uri=**https://webapp.com/callback**
&code_challenge=**F455…A353DC**
&code_challenge_method=**S256**

Authorization Code Grant
(Recommended for Most Apps)



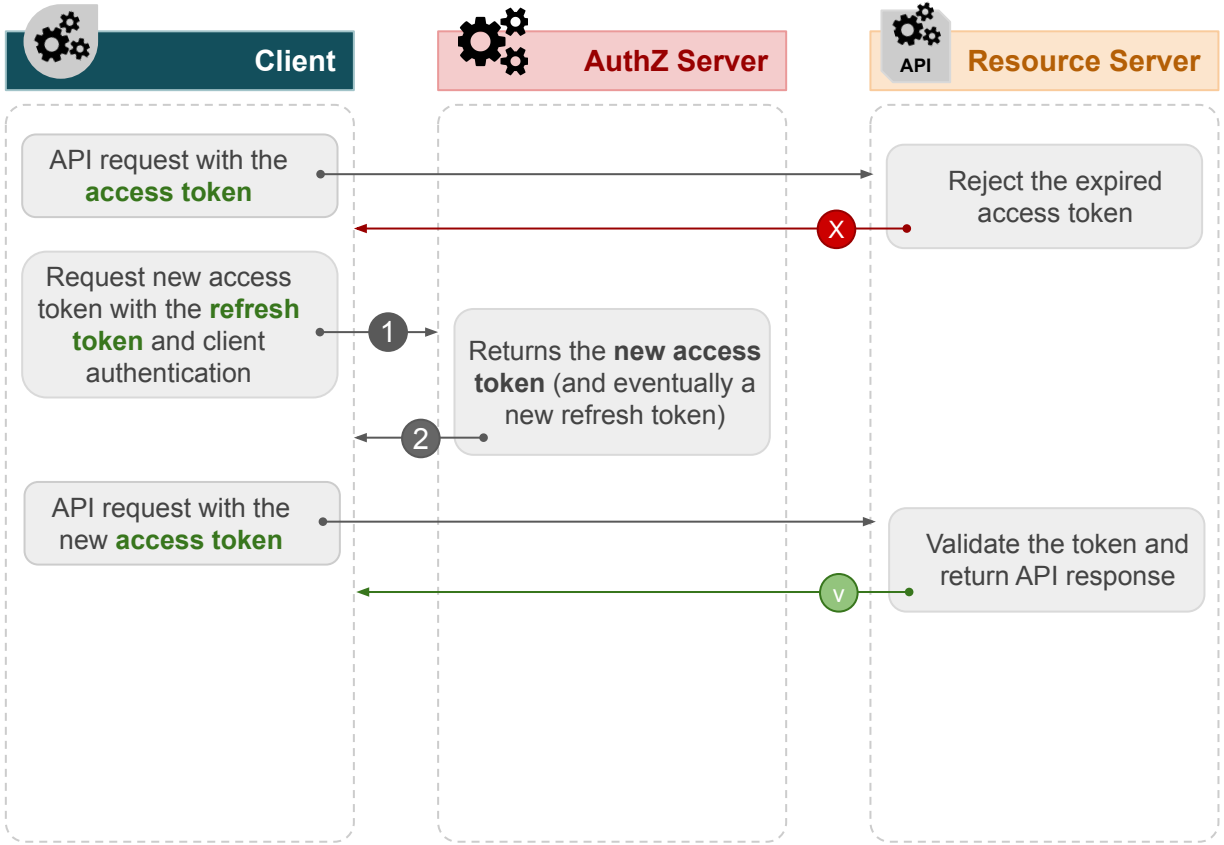
Authorization Code Grant
(Recommended for Most Apps)

Refresh Token concepts

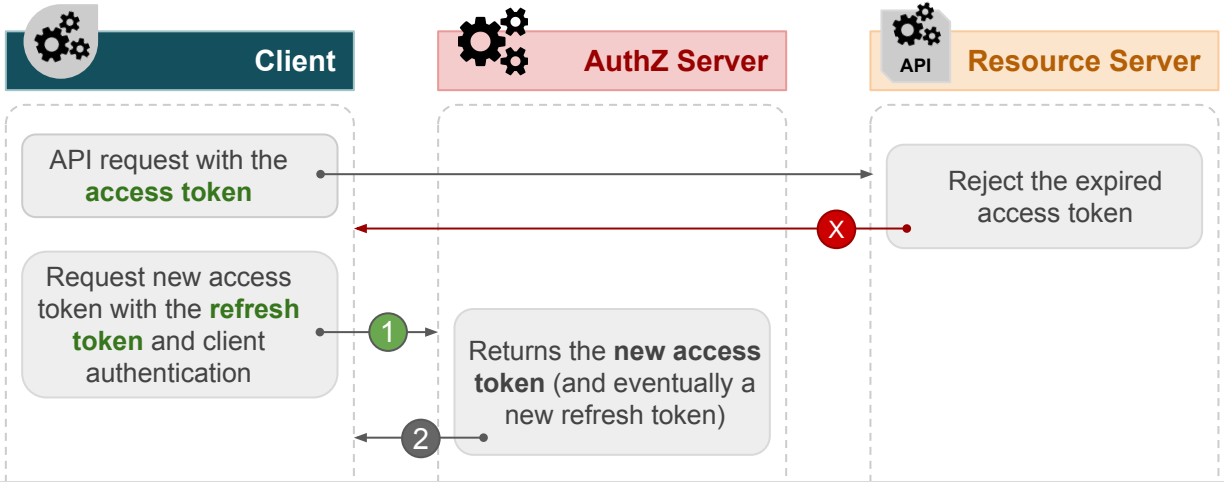
- The **refresh token lifetime** is totally decoupled from the lifetime of the authenticated user session within which the grant was performed
- Even if the expiration time associated with a refresh token is known, the client cannot assume it's always valid
  - a user could revoke consent
  - the resource server might change its local policy (e.g. accepting only access tokens obtained via multi-factor authentication from a certain moment on)
  - client code must have an appropriate error management logic
- Refresh Tokens are **Client specific**
  - i.e., a refresh token issued to Client A cannot be used by Client B
  - instead, this use case is supported by the token exchange flow

One option is to use the ***expires_in*** parameter in the token response to figure out when the access token expires.
Another option is to monitor error responses from the API when an expired token is rejected.

1. The client application decides when to use a refresh token to get new access tokens



Refresh Token Flow



1 The token authenticated request using client credential flow

POST **https://iam.cloud.infn.it/oauth/token**

grant_type=**refresh_token** → **indicates the refresh token flow**
&refresh_token=**eyJhbGciO…J9.eyJzdWEYuntNo**
&scopes=**api.read** → **this MUST be a subset of the original request**
&audience=**https://webapp.org/**

Refresh Token Flow

Refresh Token Lifetimes

- The exact refresh token lifetime is **at the discretion of the provider**
  - Refresh token lifetimes in real-world scenarios can be hours, months, or eternity
  - The provider can change its lifetime policy at will, or make it dependent on the type of client
- Refresh tokens can also be revoked
  - **Clients can revoke refresh tokens** when they no longer need them
  - Users can often revoke refresh tokens to revoke a client's authority to act on their behalf
- When a refresh token is no longer valid → re-authentication is necessary
  - The only way for the client to regain access is by **running a new Authorization Code flow**
  - This often includes explicitly requesting user involvement

Handling Refresh Tokens at the client

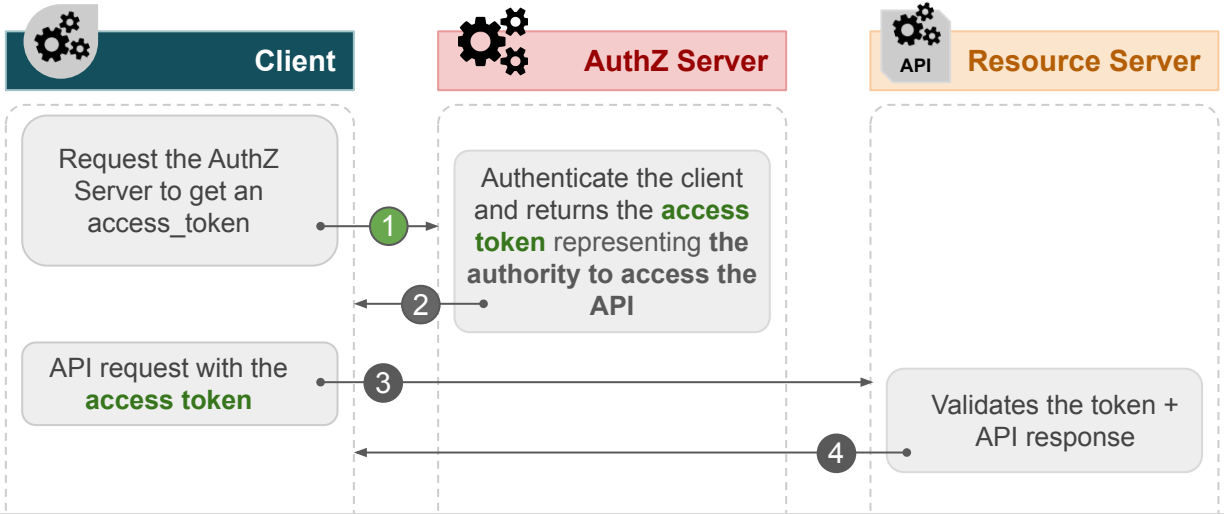
- The refresh token should be considered **as sensitive as user credentials**
  - Using the refresh token requires client authentication
- A minimum security requirement is guaranteeing confidential storage
  - This approach fails if an attacker gains access to the encrypted data and the keys

The Client Credentials Flow

The OAuth 2.0 Client Credentials Flow

- The client is accessing the API **directly**, **on its own behalf**
  - Client authentication is required (i.e. client credential flow is not enabled for public clients)
- This is an OAuth 2.0-only flow, not an OpenID Connect flow → no user involved
  - No additional authorization request is needed (i.e. **no authorize endpoint** is involved into this flow)
  - The authorization grant **does not require intervention of a user** (i.e. no login requested)
  - The **consent page is not shown** → user does not have to authorize the Client app to access its data
- The access token issued by the AuthZ Server represents the client's authority
  - the sub claim is the client unique identifier
- The Client Credentials flow only works with confidential clients
  - Confidential clients need to run in a secure environment (server-side systems)
- A Refresh Token **should not be issued** in a client credential request

RFC-6749 # Section 4.4

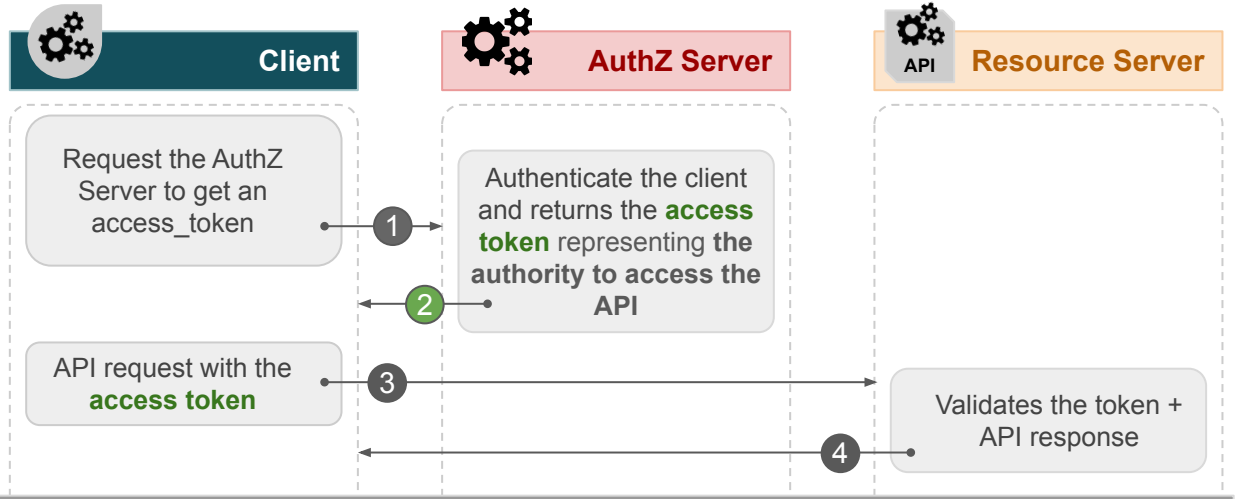


1 The token authenticated request using client credential flow

POST **https://iam.cloud.infn.it/oauth/token**

grant_type=**client_credential** → **indicates the client credentials flow**
&client_id=**901C887F-EB2E-4957**
&client_secret=**60DRv0g…0V0SWI**

Client Credentials Grant



2 The response from the Authorization Server

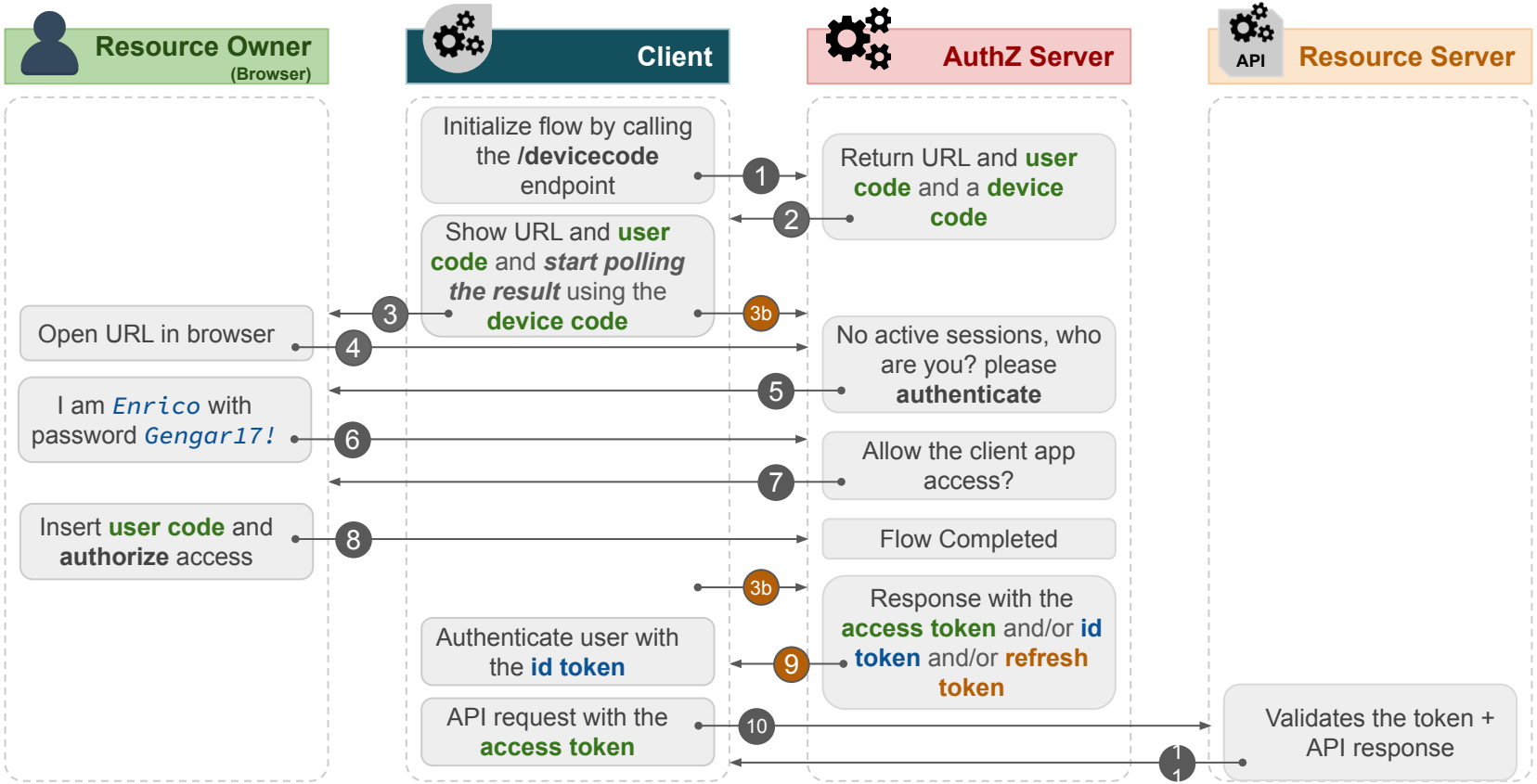
```
{
  “access_token”: “eyJhbGciOiJIUzI…zMNl0_QntNo”,
  “token_type”: “Bearer”,
  “expires_in”: 3600,
}
```

-->

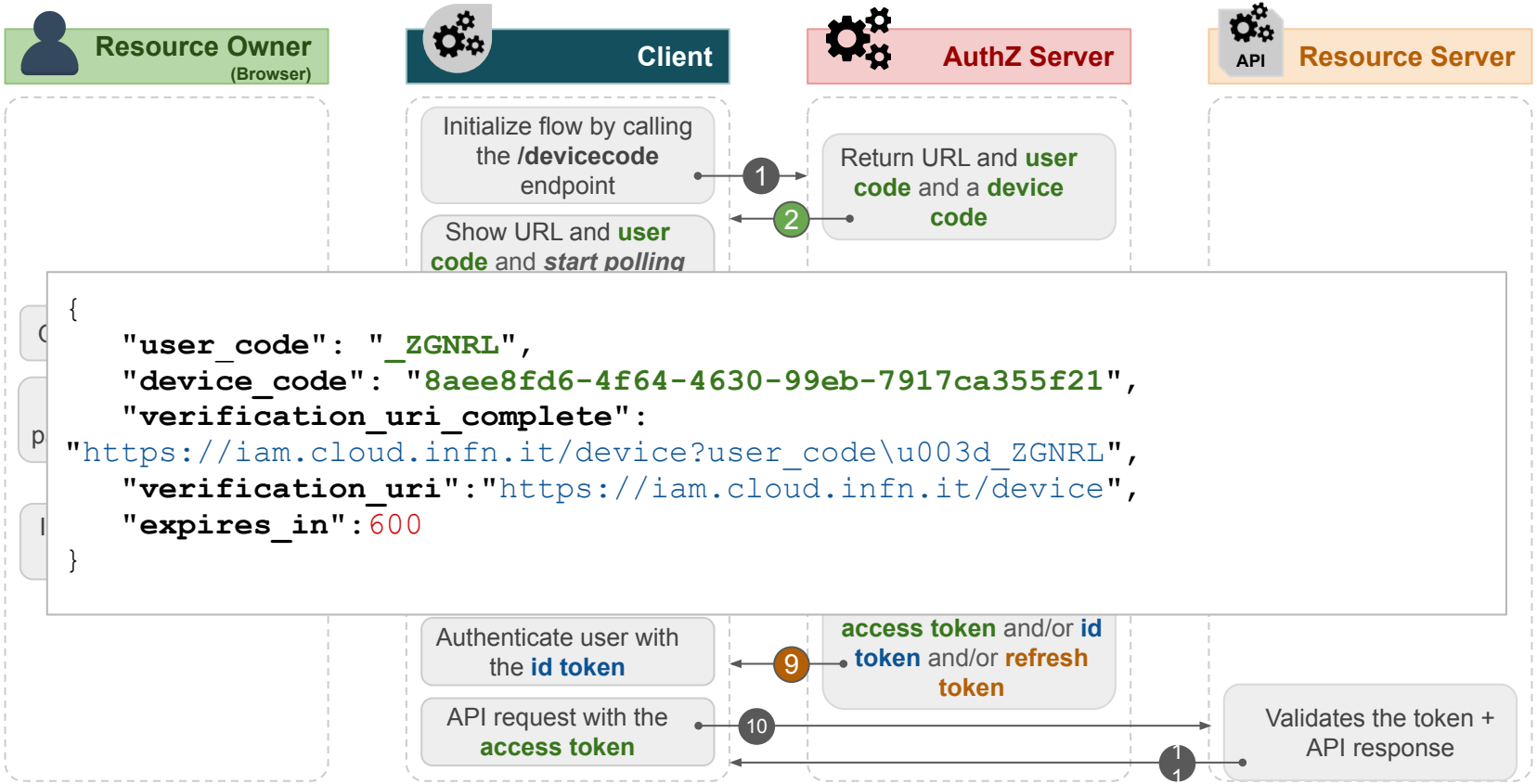
```
{
  # sub is the client id
  "sub": "901C887F-EB2E-4957",
  "scope": "api.read",
  "iss": "https://iam.infn.it/",
  "exp": 1669832797,
  "iat": 1669829197,
  "jti": "57a2e594-f212-4183"
}
```

Client Credentials Grant

The Device Code Flow



Device Code Flow



Device Code Flow

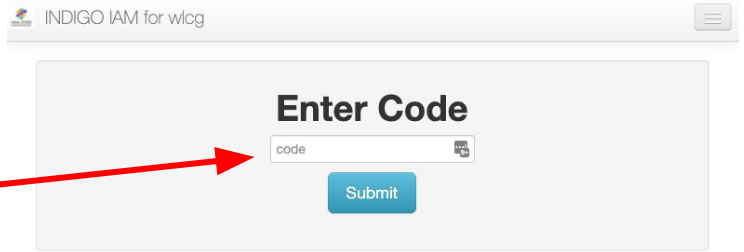
Device code flow

- RFC 8628
- Used in place of the authorization code flow **when the Client can not easily trigger a browser-based authorization**
  - the authorization to access protected resources happens on a **separate device**
- Requirements for the device code flow:
  - the device is able to display or otherwise communicate an URI and code sequence to the user
  - the user has a secondary device (*e.g.*, personal computer or smartphone) from which they can process the request
- Clients that use Device flow won't receive incoming requests that notify them about the given grant, so Clients must poll the authorization server repeatedly until user completes the approval process

Device code flow

- The authorization grant is a **code**
  - The code has to be requested at the device code endpoint exposed by the AS
  - The device code endpoint can be retrieved from the *well-known* endpoint
  - The code is used to obtain an Access Token
- Device code flow supports both public and confidential clients
- No back-channel interaction between the Client and the Authorization Server

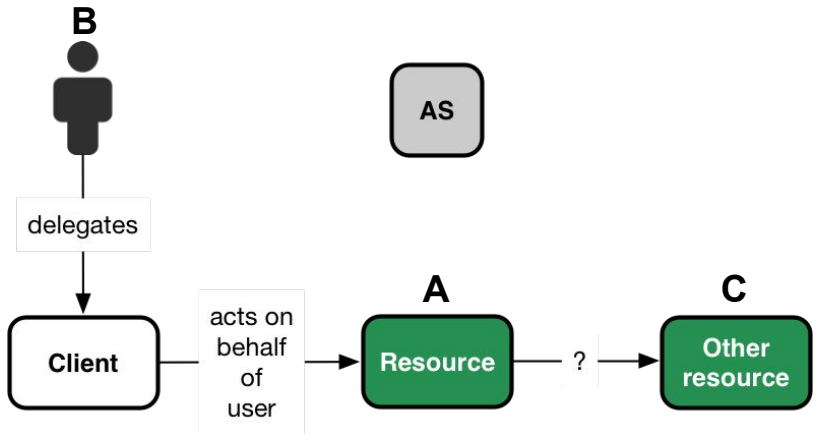
```
Registering Client ...
Generating account configuration ...
accepted

Using a browser on any device, visit:
https://wlcg.cloud.cnaf.infn.it/device

And enter the code: REUVE0
Alternatively you can use the following QR code to visit the above listed URL.
```



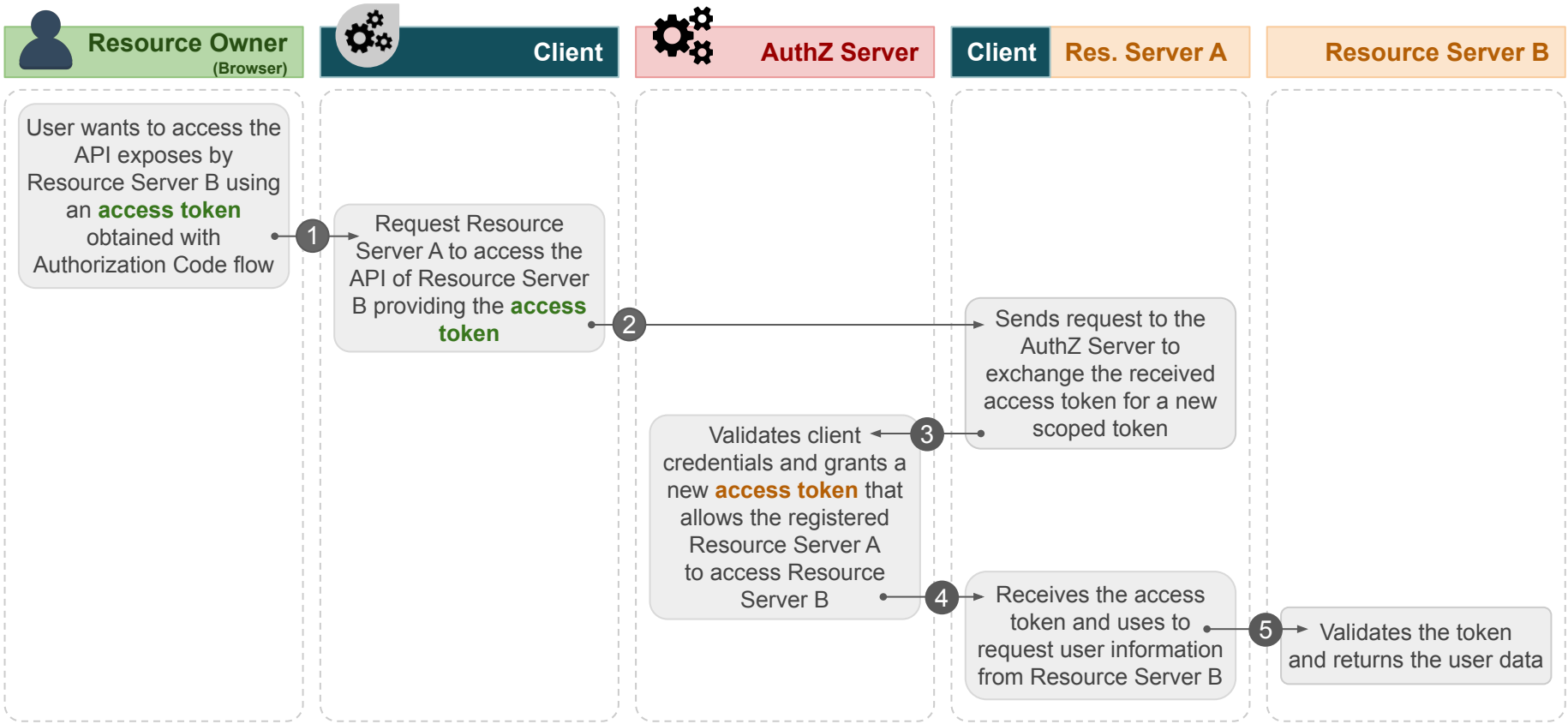
The Token Exchange flow

Token exchange

- [RFC 8693]
- This flow has been designed to satisfy the needs to access resources hosted by other downstream services on behalf of the user
    - This flow allows **R**esource **S**erver **A** to request the exchange of AT1 with AT2 (and potentially a RT2 to renew such AT2) and make calls to a backend service **C** on behalf of the requesting user **B**
    - the **R**esource **S**erver **A** is an OAuth 2 Client of the **AS**



- The exchanged token and the new token should be requested by two different Clients


Resource Owner
(Browser)
Client
AuthZ Server
Client
Res. Server A
Resource Server B
User wants to access the API exposes by Resource Server B using an access token obtained with Authorization Code flow
1
Request Resource Server A to access the API of Resource Server B providing the access token
2
Sends request to the AuthZ Server to exchange the received access token for a new scoped token
3
Validates client credentials and grants a new access token that allows the registered Resource Server A to access Resource Server B
4
Receives the access token and uses to request user information from Resource Server B
5
Validates the token and returns the user data

Token exchange concepts

- The new **access token**:
  - is more **narrowly scoped** for the Resource Server B
  - has an **audience** different from the original token (from Resource Server A to Resource Server B)
- Terminology:
  - **subject token** represents the subject access token that the Client wants to exchange
    - The act of performing a token exchange has no impact on the validity of the subject token
- Two main scenarios:
  - **impersonation** vs **delegation**
  - but "*delegation and impersonation are not inclusive of all situations. When a principal is acting directly on its own behalf, for example, neither delegation nor impersonation are in play. They are, however, the more common semantics operating for token exchange and, as such, are given more direct treatment in this specification.*" from RFC

From RFC 8693

Impersonation vs. delegation

Resource Server A impersonates Resource Owner:

- A has all the rights (determined by the scopes) of Resource Owner
- A is **indistinguishable** from B
- The process **allows a subject to change to a different subject**
  - Resource Server B cannot determine by looking at the token the identity of the Resource Server A

```
{
  "aud":"urn:example:cooperation-context",
  "iss":"https://as.example.com",
  "exp":1441913610,
  "sub":"bdc@example.net",
  "scope":"orders profile history"
}
```

Resource Owner delegates Resource Server A

- Resource Server A still **has its own identity,** which is separated from the Resource Owner one
- The optional `actor_token` used within the Token Exchange request represents Resource Server A
- When Resource Server A interacts within Resource Server B, it is **explicit** that it's representing the Resource Owner
- The Resource Owner can decide to **only delegate certain rights** to Resource Server A
- The token **act** claim in the Token Exchange response is a JSON object which identifies the acting party to whom authority has been delegated. It provides a representation of a delegation chain
  - members in the JSON object are claims that identify the actor (e.g. the *sub* claim)
  - a chain of delegation can be expressed by nesting one act claim within another: the last recent actor is the most deeply nested

```
{
  "aud":"urn:example:cooperation-context",
  "iss":"https://as.example.com",
  "exp":1441913610,
  "scope":"status feed",
  "sub":"user@example.net",
  "act":
  {
    "sub":"admin@example.net"
  }
}
```

Token exchange: use case

Example: moving some of my files with RUCIO + FTS

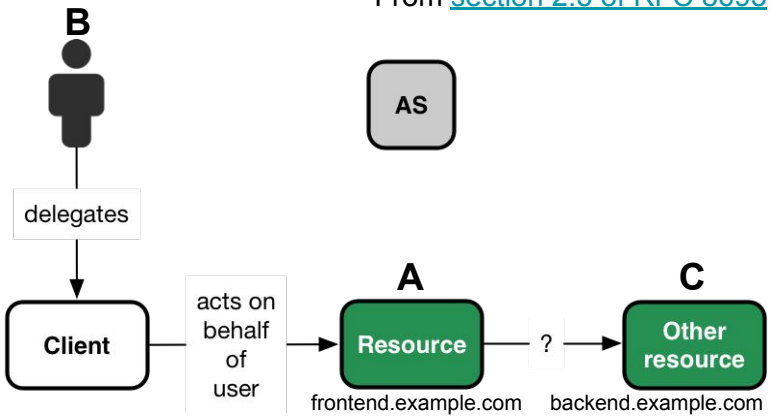
- I give RUCIO permission to act on my behalf → **Rucio registered client has token exchange grant type enabled**
- RUCIO then delegates the file transfer task to FTS, which still acts on my behalf to trigger third-party transfers across Storage Elements
    - Both RUCIO and FTS clients act on my behalf
- Different **scopes** are needed at different level of the infrastructure
- Token exchange **allows to provide tokens with minimum privileges** to each service without requiring that big fat tokens are used at the top of the chain

From WLCG CE Hackathon

The token exchange flow

From section 2.3 of RFC 8693

- User **B** wants to access the resource **C**.
- Since resource **A** is an OAuth Client of the **AS** enabled for token exchanges, B requests access to A using a bearer token issued by AS.




```
GET /resource HTTP/1.1
Host: frontend.example.com
Authorization: Bearer accVkjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC
```

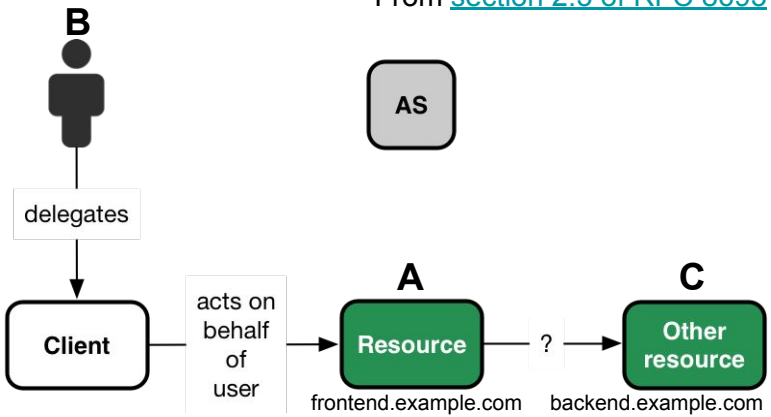
```
{
  "aud": "https://frontend.example.com",
  "iss": "https://as.example.com",
  "exp": 1441917593,
  "iat": 1441917520,
  "sub": "user-b-id",
  "scope": "resource-a-scope"
}
```

The bearer token is used by Client as grant

The token exchange flow

From section 2.3 of RFC 8693

- Then Resource A requests for a token exchange properly scoped for Resource C



HTTP basic authentication to AS using the credentials of the OAuth Client/Resource A

```
POST /as/token.oauth2 HTTP/1.1
    Host: as.example.com
    Authorization: Basic cnMwODpsb25nLXNlY3VyZS1yYW5kb20tc2VjcmV0
    Content-Type: application/x-www-form-urlencoded

    grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
    &resource=https%3A%2F%2Fbackend.example.com%2Fapi
    &subject_token=accVkjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC
    &subject_token_type=
     urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aaccess_token
```

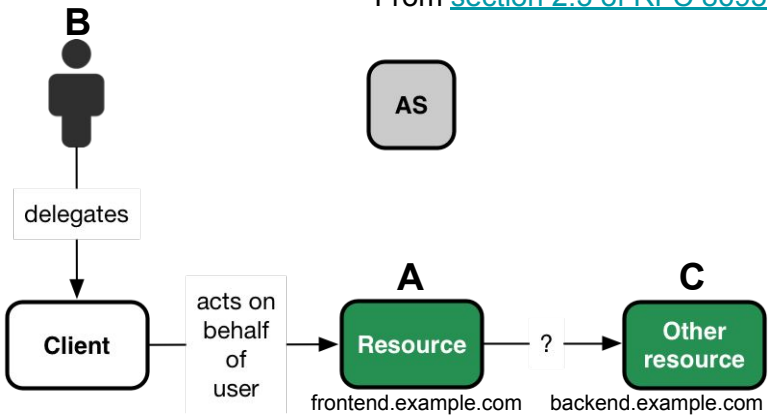
resource parameter indicates the location of the backend service (similar to *audience*)

The bearer token becomes the subject token

The token exchange flow

From section 2.3 of RFC 8693

- The AS validates both the Resource A client credentials and the subject token, and issues a new access token to Resource A (**impersonation**)




```
HTTP/1.1 200 OK
    Content-Type: application/json
    Cache-Control: no-cache, no-store

    {
     "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6Ijll
       ciJ9.eyJhdWQiO..oiYXBpIn0.40y3ZgQedw6rxf59WlwHDD9jryFOr0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCM
       y5-kdXjwhw",
     "issued_token_type": "urn:ietf:params:oauth:token-type:access_token",
     "token_type":"Bearer",
     "expires_in": 3600
    }
```

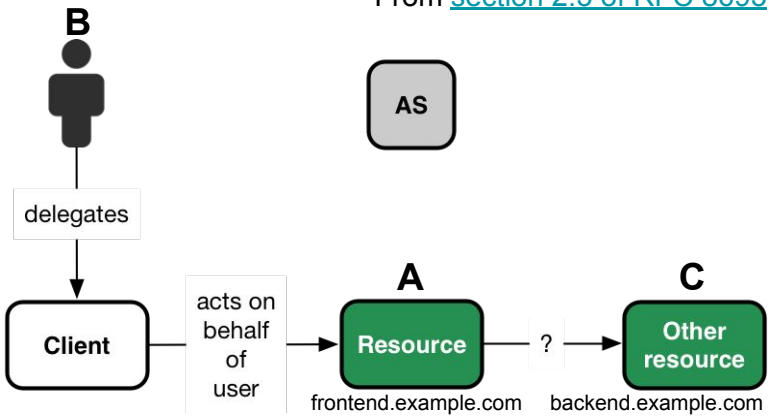

{
 "aud": "**https://backend.example.com**",
 "iss": "**https://as.example.com**",
 "exp": 1441917593,
 "iat": 1441917533,
 "sub": "**resource-a-id**",
 "scope": "**resource-c-scope**"
}

The access token is of *Bearer* type
It's opaque to the Client → it only has to be sent in another HTTP request

The token exchange flow

From section 2.3 of RFC 8693

- The AS validates both the Resource A client credentials and the subject token, and issues a new access token to Resource A (**delegation**)




```
HTTP/1.1 200 OK
    Content-Type: application/json
    Cache-Control: no-cache, no-store

    {
     "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6Ijll
       ciJ9.eyJhdWQiO..oiYXBpIn0.40y3ZgQedw6rxf59WlwHDD9jryFOr0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCM
       y5-kdXjwhw",
     "issued_token_type": "u
     "token_type":"Bearer",
     "expires_in": 3600
    }
```

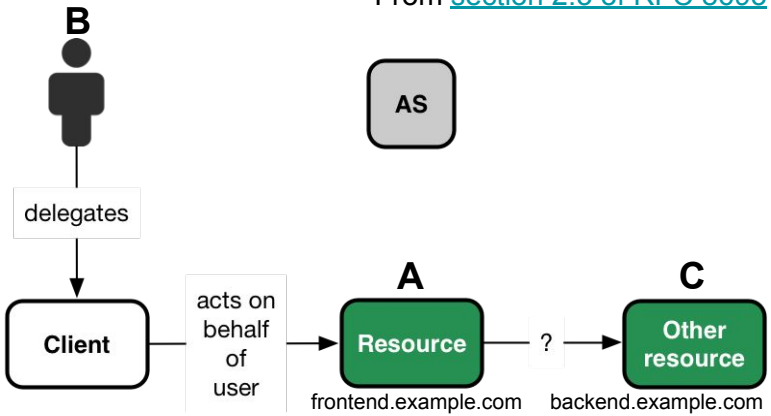
```
{
  "aud": "https://backend.example.com",
  "iss": "https://as.example.com",
  "exp": 1441917593,
  "iat": 1441917533,
  “sub”: "user-b-id"
  “act”: {
    "sub": "resource-a-id",
  },
  "scope": "resource-c-scope"
}
```

The access token is of *Bearer* type
It’s opaque to the Client → it only has to be sent in another HTTP request

The token exchange flow

From section 2.3 of RFC 8693

- Now Resource A can finally use the newly acquired Access Token to access the Resource C using HTTP bearer authentication




```
GET /api HTTP/1.1
    Host: backend.example.com
    Authorization: Bearer eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQ
       iOiJodHRwczovL2JhY2tlbmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2
       FzLmV4YW1wbGUuY29tIiwiZXhwIjoxNDQxOTE3NTkzLCJpYXQiOjE0NDE5MTc1M
       zMsInN1YiI6ImJkY0BleGFtcGxlLmNvbSIsInNjb3BlIjoiYXBpIn0.40y3ZgQe
       dw6rxf59WlwHDD9jryFOr0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIW
       vmDCMy5-kdXjwhw
```

Choosing the Right Grant Type

To select the best grant type, consider these factors:

- **Client Type** → Confidential clients (backend apps) vs. public clients (mobile, SPA).
- **Security Needs** → Use PKCE for public clients and client credentials for machine-to-machine authentication.
- **User Experience** → Use device code flow for limited-input devices and authorization code flow for web apps.
- **Use Case Mapping** → Web apps favor authorization code, while backend services use client credentials.

Dynamic Client Registration

OAuth Client registration

- Clients which interact with an Authorization Server need to be **registered**
- When a client is registered, it is assigned a unique identifier (**client_id**) and a **credential**, either
  - a password (**client_secret**), or
  - an assertion (in the form of a **JWT**)

  Credentials are required in most of the OAuth/OIDC flows or to access specific endpoints, where different privileges may be assigned to different Clients
- Client registration is necessary to integrate any application that needs to "drive" an authorization flow
  - *e.g.*, if your web app needs to authenticate users through a "Login" button, you need to register a Client

OAuth Client Types

RFC-6749 # section-2.1

confidential: clients capable of maintaining the confidentiality of their credentials (e.g., client implemented on a secure server with restricted access to the client credentials)

public: like mobile apps or single-page applications (Single Page Applications), are incapable to keep a client secret, to keep the confidentiality of their credentials

Handling client credentials

Client credentials **MUST be maintained confidential**

- not stored in Docker images or source code
  - don’t commit them on your git repo!
  - use ENV variables or other secret management mechanisms to pass down these secrets to your application

Follow recommendations in the client app security section of the OAuth security recommendations:

https://tools.ietf.org/html/rfc6819#section-5.3

Client registration on OIDC

- through the **registration endpoint** exposed by the well-known endpoint
  - i.e. **`registration_endpoint`**: `"https://iam-dev.cloud.cnaf.infn.it/iam/api/client-registration"`
- the dynamic registration can be anonymous or not
  - the OIDC provider can restrict the access to this endpoint
  - a registration access token is returned and the owner of this token can manage the client
  - these are the **dynamic clients**

Example of client registration via registration endpoint

Prepare a JSON file with the Client details, for instance

```
$ cat client_req.json
{
  "redirect_uris": [
      "https://myapp.org/callback"
  ],
  "client_name": "client-demo",
  "contacts": [
      "enrico.vianello@cnaf.infn.it"
  ],
  "token_endpoint_auth_method": "client_secret_basic",
  "scope": "address phone openid email profile offline_access",
  "grant_types": [
      "refresh_token",
      "authorization_code"
  ],
  "response_types": [
      "code"
  ]
}
```

Example of client registration response

```
$ curl https://iam-dev.cloud.cnaf.infn.it/iam/api/client-registration -H "Content-Type: application/json" -d @client_req.json 2>/dev/null
| jq
{
  "client_id": "90b4f677-2551-4852-935e-8f785c583572",
  "client_secret": "xxx",
  "client_name": "client-demo",
  "redirect_uris": [
        "https://myapp.org/callback"
  ],
  "contacts": [
        "enrico.vianello@cnaf.infn.it"
  ],
  "grant_types": [
        "authorization_code",
        "refresh_token"
  ],
  "response_types": [
        "code"
  ],
  "token_endpoint_auth_method": "client_secret_basic",
  "scope": "openid profile offline_access email",
  "reuse_refresh_token": true,
  "dynamically_registered": true,
  "clear_access_tokens_on_refresh": true,
  "require_auth_time": false,
  "registration_access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3O…TE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c",
  "registration_client_uri": "https://iam-dev.cloud.cnaf.infn.it/iam/api/client-registration/90b4f677-2551-4852-935e-8f785c583572",
  "created_at": 1669116921824
}
```

towards OAuth 2.1

What's new in OAuth2.1

- OAuth 2.1 (draft)
  - it is a draft with the aim of consolidating and simplifying the most commonly used features of OAuth 2.0
- Some new features of OAuth 2.1
  - PKCE (Proof Key for Code Exchange) **is required for all OAuth Clients using the authorization code flow**
  - redirect URIs must be compared using **exact string matching** → no wildcards in the URI
  - the **implicit grant is omitted** from this specification
  - the **resource owner password grant is omitted** from this specification
  - using bearer tokens in the query string of URIs is forbidden

that’s all

Useful references

RFC

- The OAuth 2.0 Authorization Framework (6749)
- JWT (7519)
- Bearer token usage (6750)
- OAuth 2.0 Device Authorization Grant (8628)
- Token exchange (8693)
- Proof Key for Code Exchange (7636)
- JWT for client authentication (7523)
- OpenID Connect 1.0

Draft

- The OAuth 2.1 Authorization Framework
- OpenID Connect federation

Other

- OAuth 2.0 and OpenID Connect video (OktaDev)

Part 2: Hands-on

Now it’s your turn to make practice

- Device Code Flow
- Client Credentials Flow
- Refresh Flow
- Token Exchange Flow

Backup

Authorization code flow

- Section 4.1 of RFC 6749 (OAuth 2)
- Section 3.1 of the OpenID Connect spec
- It’s the recommended flow for server-side applications that can maintain the confidentiality of client credentials
    - recommended also for any client when combined with PKCE

Authorization code flow

OAuth 2.0 and OpenID Connect video from OktaDev