The missing piece in MCP Authorization

Or:
“Demistifying MCP Authorization” ?
By Alex Babeanu

Authorization, in general, is confusing, especially for those who don’t usually evolve in Identity and Access management (IAM) circles. And then there’s Model Context Protocol (MCP) Authorization, which can be extremely confusing for developers not privy to the intricacies of the various OAuth2 standards involved. Let’s try to make this part of the MCP Specification a little clearer, and identify what’s missing in it…

Contents:

OAuth2 roles in MCP

MCP adopted OAuth2 as its standard for authorization. OAuth2 uses standardized processing flows that usually involve 4 actors : the Subject, the Client, the Authorization Server and the Resource Server. These roles can be a major source of confusion, as the same actual system can behave differently from one use-case to another. We thus first need to properly define the OAuth2 roles of each of the actors in the MCP system…

OAuth2 Role	Actor(s)	Description
Subject	End-user, or Agent	The human initiating the requests. Note that in the case of Autonomous Agents, the Agents themselves may be the Subjects (see below).
Client	MCP Client	The App or system that uses the MCP Server's Tools or Resources. The client typically includes an AI Model component (running locally or remotely), which decides which Tools to use.
Resource Server (RS)	MCP Server	The MCP Server itself. The Resources to protect are the Tools, Resources or Prompts exposed by the MCP Server. In Authorization terms, these are all *Resources,* access to which must be authorized.
Authorization Server (AS)	An external or third-party Identity Provider	An external or third-party system or service that can authenticate all actors, and issue OAuth2-compliant access tokens.

About OAuth Subjects

Determining the proper Subject of an MCP request can be tricky, because as stated, both the prompting end-user OR the AI Model itself could be the actual Actor invoking the MCP Tools. For instance, it is possible to imagine use-cases where the Agent doesn't react to prompts or API calls directly initiated by human actors, but instead reacts autonomously to certain events or to environmental changes. Consider, for example, an Agent trained to read logs files, and reacting in real-time to some concerning events it finds there. That agent could then autonomously trigger notifications and/or remediation actions by selecting the right set(s) of tools to use. In those cases the actor, the Agent itself, is the Subject for all authorization intents and purposes.

Nevertheless, in certain verticals or regulated environments (banks for example), the responsibility of an Agent's action should always fall on a human. In those cases, a default

“owner” for the agent can be assigned, and all potential remediations for an Agent’s action can then be tied back to that owner.

In any case, the Subject will be identified by the `sub` claim in the `access_token` presented for any MCP Tool use. The Subject is the entity which access entitlements will be evaluated during the Authorization step (see further below).

The MCP OAuth2 Authorization flow

Registration

For any of these flows to work, all actors must be known, i.e., registered with both the AS and any other system protecting the RS. This could be difficult in the cases where the MCP server is exposed to the Internet and can accept client requests from any external domain. In such cases, Just-In-Time provisioning (JIT) of the Subjects can be implemented, if the AS supports it, as well as dynamic Client registration (through the RFC 7591 specification). These are made explicit in the overall flow section below.

It is also possible (and more likely in Enterprise scenarios) that Subjects and Clients are all pre-registered with all systems involved via out-of-band processes.

High-Level Description

The Access Token

The MCP specification relies on the 3-legged `authorization_code` OAuth2 flow. The MCP Client first requests a Code from the AS, then exchanges the code for a proper `access_token`. Getting the token therefore requires the Client to make two calls to the AS:

1. Request a code, by first authenticating itself with the AS as the Client. This step enables the AS to optionally communicate directly with the human Subject to get an optional human approval, if necessary. Then,

2. Exchanging the code for a proper `access_token`.

But there is a problem. If the MCP Clients and Servers are in different Trust domains across the Internet, and don’t really “know” each other, then how does the MCP Client know which AS to contact to get a right `access_token` to use for the Tool it tries to access?

To solve this problem, the MCP specification requires all parties to use standard OAuth2 **Metadata documents**:

- RFC 9728 specifies the **Protected Resource Metadata Document**. This document describes the various configurations supported by the RS (the MCP server), and in particular provides the list of Authorization Servers that can be used with it.

- RFC 8414 defines the **AS Metadata Document**. It lists the URL Endpoints of the AS for all the OAuth2 flows it supports.

These two documents enable MCP Clients to discover dynamically at runtime all the details pertaining to the AS, in particular which AS to use and the list of its OAuth2 endpoint URLs.

The missing piece: Dynamic Authorization and AuthZEN

Thus, the OAuth2 `authorization_code` flow enables the following things:

- Authenticate the MCP Client
- Authenticate the Subject
- Optionally request an approval from the User Subject
- Optionally let the AS augment the `access_token` with claims, such as `scope`, that can then be used by the RS to further restrict access to its resources.

Nevertheless the OAuth2 flows were created for human interactions and static Clients and RS's. They were not made at all to support highly dynamic Agentic AI environments. MCP Clients can utilize any Remote MCP Servers, it is therefore impossible for an AS protecting a set of MCP Servers to know what a newly registered MCP Client should have access to. New MCP Clients can only be granted default, high-level access rights… These coarse entitlements may not be sufficient if the MCP tools grant access to sensitive backend resources, such as documents or data tables. Protecting these sensitive backend resources requires fine grained authorization, which must be done at the RS level. **This piece is completely missing in the MCP Specification**… For this, we need OpenID AuthZEN 1.0, as detailed below.

PEP / PDP architecture

To understand AuthZEN, we need to look at the concepts of Policy Enforcement Point (PEP) and Policy Decision Point (PDP), which emerged in the early 2000's with the XACML specification, and were formalized further in NIST's Zero Trust reference architecture (NIST SP 800-207).

The PEP/PDP architecture is the basis of all real-time dynamic authorization and Zero Trust, and the only way to implement its "Never trust, Always verify" precept.

Instead of performing authorization once at authentication time, like in the OAuth2 flows, every single request for a resource must be verified, every single time.

To achieve this, Resource Server implementers must build or use a PEP. The PEP is a proxy, API Gateway or simple `if / then/ else` code block that redirects all incoming requests to a PDP for evaluation.

Figure 1 below shows the usual high-level MCP interaction between the Subject, Client and MCP Server. The Server hosts a number of tools that the MCP Client tries to access:

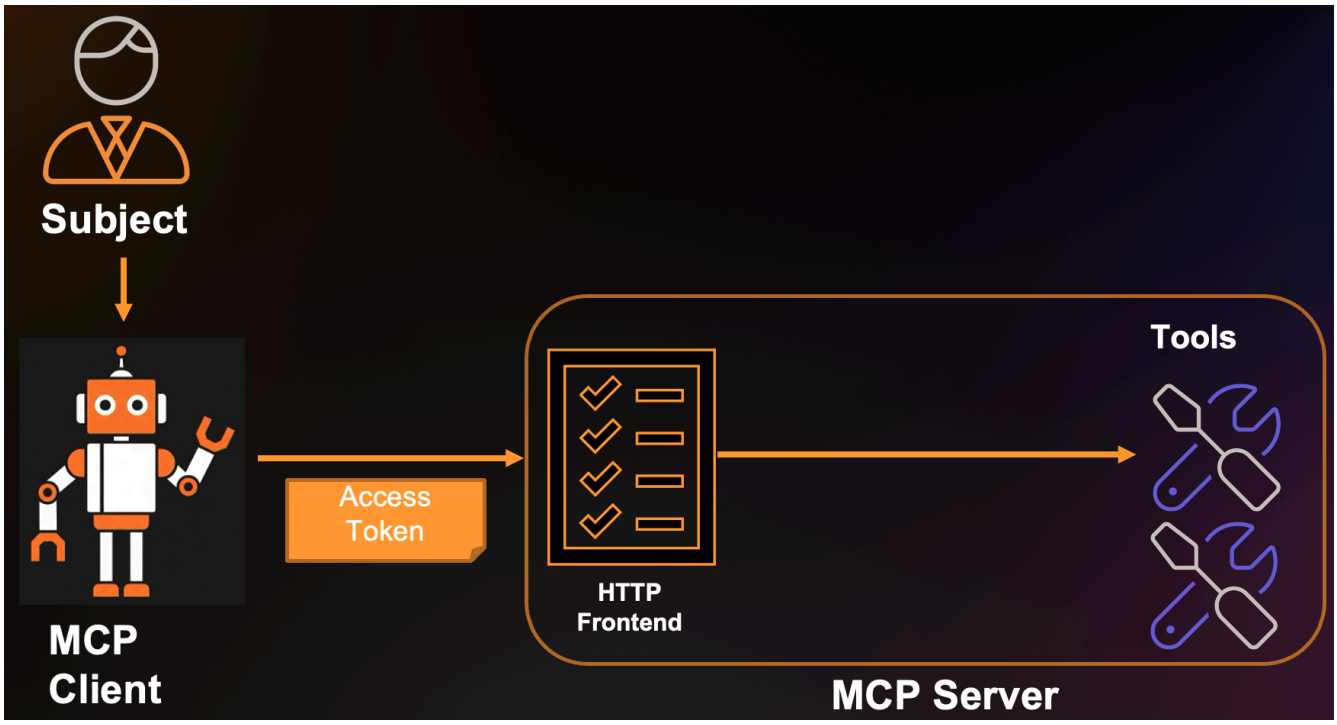

Figure 1: MCP interactions and Tools

The idea is to insert a PEP in front of the Tools, as depicted in Figure 2 below; i.e.,

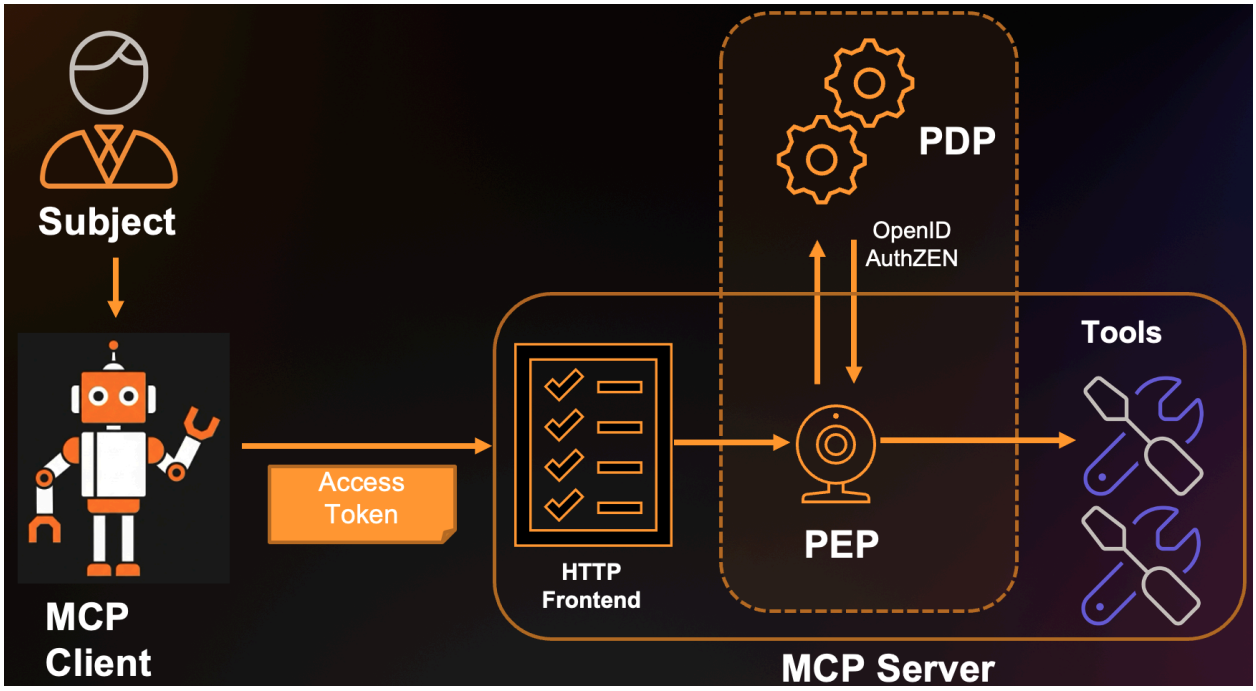

Figure 2: PEP and PDP placement in MCP Servers

- The PEP must be inserted "in front of" every MCP Server Tool or Resource. That is, a code wrapper must be added to catch all incoming requests and evaluate them before actually executing the Tool or Resource functions.
- The role of the PEP is to redirect all such requests to the Policy Decision Point (PDP). The PDP is a rules engine programmed with deterministic **Access Policies**. These can be more or less complex conditions that are expressed using some Authorization Model, such as ABAC, ReBAC, RBAC or other. The PDP will evaluate the requests against these rules and Grant or Deny access to the tools.
- The Responses are returned to the PEP, which then lets the requests proceed or returns an `Unauthorized` error.
- The communication protocol between PEP and PDP is defined in the OpenID **AuthZEN** specification.
- AuthZEN enables the MCP Server to use any AuthZEN-compliant external PDP.
- Many AuthZEN compliant off-the-shelf solutions exist (such as Indykite), and should be used.

Externalizing authorization enables central management of policies, compliance to laws and regulations and enables administrators to define very fine grained authorization for the MCP Tools. This architecture constitutes best practices and the only way, really, to implement Zero-Trust precepts.

Putting it all together: full MCP authorization flow

Given all these, the full MCP Authorization flow is as follows (refer to the Authorization Flow Steps diagram of the MCP Specification as a reference):

1. The User accesses the MCP Client somehow (via prompt, API call other).

Note: MCP does not specify how the User authenticates with the MCP Client, that part is left at the discretion of the MCP Client implementers.

2. The MCP Client's AI model determines it must use some tool(s) in order to accomplish the goal set by the Subject user.

3. The MCP Client makes the corresponding MCP request to the MCP Server. This initial request contains no token at all.

4. The MCP Server sees no `access_token` in the incoming request, and therefore returns an *HTTP 401 Unauthorized* error with a `WWW-Authenticate` Header. The header will contain a `resource_metadata` Bearer header that points to the location of the MCP Server's **Protected Resource Metadata Document**.

E.g., Unauthorized response from MCP Server:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer resource_metadata=
"https://mcp.example.com/mcp/v1/<projectID>/.well-known/oauth-protected-resource
```

5. The MCP Client extracts the `resource_metadata` from the response header, then GETs that document:

```
GET  https://mcp.example.com/mcp/v1/<projectID>/.well-known/oauth-protected-resource
HTTP/1.1   Host: mcp.example.com
```

The response will look something like this:

```
{
  "resource": "https://mcp.example.com",
  "authorization_servers": [
    "https://as1.example.com",
    "https://as2.example.net"
  ],
  "bearer_methods_supported": [
    "header"
  ],
  "scopes_supported": [
```

```
        "profile",
        "email",
        "phone"
      ],
      "resource_documentation":
    "https://mcp.example.com/resource_documentation.html"
    }
```

6. The MCP Client must then validate this returned metadata. Validation is described in section 3.3 of RFC 9728 but in essence, the value of the `"resource"` response property must be the same as the URL of the MCP Server the Client is accessing. Optionally the metadata can be signed by the MCP Server. In that case, the signature must also be validated by the Client (this means the Client and Server “knew” each other beforehand, and exchanged certificates).

7. The MCP Client chooses an AS to use from the provided `"authorization_servers"` list, and fetches the AS Metadata Document from there:

```
GET https://as1.example.com/.well-known/oauth-authorization-server HTTP/1.1
    Host: example.com
```

The response from the AS will be similar to the example below:

```
HTTP/1.1 200 OK
Content-Type: application/json
     {
      "issuer":
        "https://as1.example.com",
      "authorization_endpoint":
        "https://as1.example.com/authorize",
      "token_endpoint":
        "https://as1.example.com/token",
      "token_endpoint_auth_methods_supported":
        ["client_secret_basic", "private_key_jwt"],
      "token_endpoint_auth_signing_alg_values_supported":
        ["RS256", "ES256"],
      "userinfo_endpoint":
        "https://as1.example.com/userinfo",
      "jwks_uri":
        "https://as1.example.com/jwks.json",
      "registration_endpoint":
        "https://as1.example.com/register",
      "scopes_supported":
        ["openid", "profile", "email", "address",
         "phone", "offline_access"],
```

```
  "response_types_supported":
    ["code", "code token"],
  "service_documentation":
    "http://as1.example.com/service_documentation.html",
  "ui_locales_supported":
    ["en-US", "en-GB", "en-CA", "fr-FR", "fr-CA"]
}
```

The MCP Client now knows both the `"authorization_endpoint"` and `"token_endpoint"` to use to get an `access_token` from this AS.

8. The MCP Client initiates an OAuth2 flow using the AS's `"authorization_endpoint"`. This request MUST include a **resource** parameter specifying the URL of the requested tool(s). For example, using PKCE (note: always use PKCE), requesting access to two MCP tools :

```
HTTP/1.1
  GET https://server.example.com/authorize?response_type=code
     &client_id=s6BhdRkqt3
     &state=tNwzQ87pC6llebpmac_IDeeq-mCR2wLDYljHUZUAWuI
     &redirect_uri=https%3A%2F%2Fclient.example.org%2Fmcp
     &scope=tools%3Atool1%20tools%3Atool2
     &resource=https%3A%2F%2Fmcp.example.com%2Ftools%2Ftool1
     &resource=https%3A%2F%mcp.example.com%2Ftools%2Ftool2
  Host: authorization-server.example.com
```

The AS will then proceed to:

a. (Optionally perform dynamic Client Registration, if the MCP Client was not yet known.)
b. Authenticate the MCP Client.
c. Authenticate the Subject. This will likely involve a back-and-forth flow with the user-agent (their browser, for example), to request the user's credentials. This step may use multi-factor authentication.
d. (Optionally auto-provision the Subject, if not already known)
e. (Optionally request the User's approval for the MCP Client to use the requested MCP Server resource.)
f. Return an `authorization_code` to the MCP Client.

9. The MCP Client can then package the call to exchange the code for the `access_token`:

```
HTTP/1.1
  POST https://as1.example.com/token   Host: as1.example.com
  Authorization: Basic czZCaGRSa3F0Mzpoc3FFelFsVW9IQUU5cHg0RlNyNHlJ
```

```
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fmcp
&code=10esc29BWC2qZB0acc9v8zAv9ltc2pko105tQauZ
&resource=https%3A%2F%2Fmcp.example.com%2Ftools%2Ftool1
&resource=https%3A%2F%2Fmcp.example.com%2Ftools%2Ftool2
```

The MCP client will then receive the `access_token` with the requested scopes and resources. In this example, the scopes grant access to `tool1` and `tool2` of the MCP Server. These scopes grant coarse/high level only access to the tools.

Note: that the access token's `audience` claim will hold the value(s) of the resources requested in step 8 above. The token will therefore only be usable there. Optionally, the AS may also have augmented the token with additional custom claims, depending on the implementation. In any case, the response should be as follows:

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

 {
     "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6Ijc3In0.eyJpc3MiOiJodHRwOi8vYXMxLmV
     4YW1wbGUuY29tIiwic3ViIjoiX19iX2MiLCJleHAiOjE1ODg0MjA4MDAsInNjb3BlIjoibWNwOnRvb2
     wxIG1jcDp0b29sMiIsImF1ZCI6Imh0dHBzOi8vbWNwLmV4YW1wbGUuY29tLyJ9.nNWJ2dXSxaDRdMUK
     lzs-cYIj8MDoM6Gy7pf_sKrLGsAFf1C2bDhB60DQfW1DZL5npdko1_Mmk5sUfzkiQNVpYw",
    "token_type":"Bearer",
    "expires_in":3600,
    "refresh_token":"4LTC8lb0acc6Oy4esc1Nk9BWC0imAwH7kic16BDC2",
    "Scope":"mcp:tool1 mcp:tool2"
 }
```

10. The MCP Client can now use `access_token` to invoke either `tool1` or `tool2`.

11. Nevertheless these tools may potentially grant access to confidential or sensitive data from their backend. A further, fine-grained authorization step is therefore necessary. The MCP server tool, now acting as a **PEP**, packages an AuthZEN Request to an external PDP (such as Indykite). For example, the `tool1` function can access some file system documents in the backend service, some of which may be sensitive and not presentable to all Subjects. It therefore needs to fetch only those documents that the Subject can actually access. `tool1` thus performs an AuthZEN Resource Search request:

I.e., "Which documents can Subject Alice read?" :

```
{
```

```
  "subject": {
    "type": "user",
    "id": "alice@example.com"
  },
  "action": {
    "name": "can_read"
  },
  "resource": {
    "type": "Document"
  }
}
```

The response may be as follows:

```
{
  "results": [
    {
      "type": "Document",
      "id": "123"
    },
    {
      "type": "Document",
      "id": "456"
    }
  ]
}
```

12. The tool can now package its response to the MCP Client, the whole flow, down to the fine data elements has been authorized.

Drafts / Notes / cut-off bits - DO NOT PUBLISH:

About Authorization

The term “Authorization” itself can be the source of this confusion in MCP and elsewhere. It really includes two distinct concerns that should be separated: Authentication and Authorization proper, typically, and unfortunately, done at the same time in OAuth environments.

Authentication (AuthN) is the process by which a service or server asserts the identity of the Subject or Principal that tries to access its functionality.

Authorization (AuthZ) on the other hand, happens after Authentication, once the identity of the subject has been established without a doubt. It tries to assert the entitlements, or rights, that the given subject has on the resource they are trying to access.

Both AuthN and AuthZ are necessary in order to properly protect any given system, or “Resource”, including MCP tools. But where AuthN is considered for the most part a solved problem, AuthZ on the other hand is a difficult nut to crack. Every system has different resources it needs to protect, and there is no one-size-fits all set rules that could possibly work for all. AuthZ therefore typically requires implementing and evaluating a set of Access Policies at runtime in order to determine access.

About MCP clients

Given these actors, the MCP Client is often an AI model powered App. The difficulty in using OAuth with MCP arises from the fact that OAuth2 was created for human-centered static Clients, built specifically for accessing certain resources in a rather static way. MCP Clients being AI-powered are non-deterministic. It is rather impossible to predict which tools in which Trust Domain an MCP Client will try to use.

The Problem

The main source of confusion so far for developers, is the fact that there has really only been one single family of standards to solve both the AuthN and AuthZ problems for humans over HTTP: the OAuth 2.x family (of which there are over 100 RFCs !). It’s an ecosystem of standards that is hard to navigate indeed…The complexity is such that Broken Access Control is still the #1 Web Application security issue right now.

The problem then, is that using OAuth 2, AuthN and AuthZ happen *both at the same time*: at the time when the `access_token` is minted by the "Authorization Server" (the AS).

The first issue with this approach is that once a Subject or Client has possession of an access token, they can keep using it until it expires, regardless of whether the data it contains (the claims) are still valid or not. If the Subject is a user, better User Experience requires long-lived tokens (to avoid incessant user logins for instance), such tokens can therefore end-up granting more access than they should. This very problem is the basis of the CAEP profile for the OpenID Shared Signals specification (SSF), which aims to keep those session tokens up-to-date when access changing events occur anywhere in the system. Adding CAEP and SSF on top of Oauth becomes a rather big complication, more than most developers are willing to chew.

The second issue with the approach is token size. At the time of authentication the AS knows very little about the intentions that Alice has on the resource. In fact, given that the Client is essentially an AI Model, nobody can possibly know what the Client will attempt.. The AS may thus need to add all the possible entitlements that Alice has on the resource as Scope claims, for example, in the access token. Since these tokens are passed as request headers, they are therefore subject to the header size limitations that are enforced by most Web Servers. This will necessarily limit the amount of data that can be stored in those tokens.

Full details of the OAuth2 flows can be found here for example: https://oauth.net/2/,

Anthropic has selected OAuth 2.1 as its "Authorization" standard for MCP (well, really for both AuthN and AuthZ, our problem), even though most of the IAM industry recognizes that it is not the right standard for the task. But none other existed, so here we are…

How MCP

Consider the following typical example:

- A developer builds an API for some business goal, and deploys it. For example: accessing Employee records.
- Another developer, who may or may not work for the same organization, decides to use the API for their own purposes. For example, to provide the employees with a set of benefits.

Protecting this Employee Record API requires the first developer to:

- Identify the actor that tries to access their service: is it a known user?

- Determine if that Subject is entitled to not only access the API, but also whether they are entitled to perform the requested action of the given data points or backend system. This evaluation can be quite granular: if for example Alice can access her own Employee Record, she shouldn't be able to access her manager's. In both cases, Alice would use the same GET Employee API…

How it's done with OAuth2, in brief (for full details, refer to this site for example:) :

- The End-User, Alice, is the **Subject**.
- Alice Registers for her employer's benefits program, in the Benefits App. This is the **Client**.app.
- The Client needs to read Alice's Employee Record, and must use the Employee Record API for that. That APi is the **Resource**.
- In order to make this happen, both Client and Resource must use the same **AS**.
- When Alice accesses the Client, the client redirects Alice to the AS for authentication, at which point she (her browser really) gets an access token.
- After the authentication process (using whatever factors have been configured), the AS can add additional claims to the token it creates for Alice. These additional claims can be **Scopes**, identifying things that Alice can do on the resource, and/or can be custom values, such as Alice's Role, department, location, etc.. This is optional of course, but this token augmentation is typical.
- The Client can then access the API, by identifying itself (with an API key, another Client token or other means), and by passing-on Alice's token.
- The Resource service, the Records API, can then validate the token with the AS and grant access (or not).

The problem here is that the authorization evaluation is done only once: at the time when the access token is minted, which typically happens when the Subject is authenticated.

If anything changes in Alice's context (her job, role, or if she gets terminated for example), the access token still contains all those now invalid claims. This could potentially grant Alice access she shouldn't have anymore; a serious Access Control issue!
Besides,

Authorization Best Practices

This is where we need to have a closer look at the standards, and include the latest ones too…