SECUREAUTH
Welcome to Better Identity.

HOT TOPIC

AGENTIC AI

IDENTITY 101 FOR AI AGENTS

AI agents are becoming trusted actors in enterprise systems—but trust demands identity. This piece breaks down how OAuth 2.1 and OpenID Connect provide the foundation for secure, non-interactive authentication and authorization, enabling autonomous agents to operate safely at scale across modern digital environments.

ARCHITECTING IDENTITY FOR AGENTIC AI: PART 1

This is **Part 1** in the series **Architecting Identity for Agentic AI**, a technical guide for software architects securing AI systems with OAuth, OIDC, and trust frameworks. Designed for architects building scalable, policy-driven identity across enterprise and cross-boundary environments.

BY LUKASZ RADOSZ

SVP of Engineering, SecureAuth

With over two decades in identity and access management, Lukasz brings deep expertise in authentication, authorization, and API security to the SecureAuth team. A champion of open standards like OAuth and OIDC, he's an advocate for modern identity architectures across machine identity, open banking, and transactional access control.

Introduction

Agentic AI systems—software agents with autonomous or semi-autonomous capabilities—are rapidly emerging in enterprise environments. As these AI agents interact with various services and APIs, establishing a secure identity for each agent is foundational. Just as human users have identities to authenticate and authorize their actions in a system, AI agents require machine identities to prove who (or what) they are and what they are allowed to do.

This article (Part 1 of a progressive series on AI agent identity) lays the groundwork for understanding how modern identity protocols apply to AI agents. We will introduce core concepts from OAuth 2.1 and OpenID Connect (OIDC), including the token flows and token types that are most relevant to AI agents. By the end, you should grasp how an AI agent can securely obtain tokens to access resources, and how these processes differ from human-driven identity flows.

Future parts in the series will build on this foundation, exploring trust frameworks, token exchange, OpenID federation, and dynamic authorization for AI agents.

Identity for AI Agents in the Enterprise

In an enterprise setting, **AI agents** (such as intelligent assistants, automated scripts, or services using AI) often need to call protected APIs and services. To do so securely, they must present credentials and tokens just like a human user would. However, unlike human users, agents don't log in with a username/password or MFA prompts—their authentication must be automated and **non-interactive**. This introduces the need for **machine identity flows** that parallel human SSO (Single Sign-On) flows but eliminate any manual steps.

Key considerations for AI agent identity include:

- **Authentication**: Verifying the agent's identity (e.g. using a client ID and secret, certificate, or key) without human involvement
- **Authorization**: Ensuring the agent only accesses what it's permitted to (via scopes or permissions encoded in tokens)
- **Audit and Management**: Treating agent identities as first-class entities—manageable, auditable, and revocable by the organization's identity management system, similar to user accounts

In practice, enterprises often handle agent identities through mechanisms like service accounts or registered OAuth clients. An AI agent might be registered as an OAuth client with the corporate Identity Provider (IdP). This allows the agent to use standardized flows to obtain access tokens that grant it limited access to enterprise APIs. The challenge is to do this in a secure way, aligning with industry standards. This is where OAuth 2.1 and OpenID Connect come in.

OAuth 2.1 Fundamentals for Secure Access

OAuth 2.1 is the latest evolution of the OAuth authorization framework, consolidating the security best practices learned from OAuth 2.0. OAuth's primary goal is **delegated authorization:** enabling one entity (like our AI agent) to access resources on behalf of itself or a user, without handling long-lived credentials like passwords. For AI agents, OAuth provides a way to get time-limited tokens to act within approved bounds.

Core OAuth Roles: In any OAuth scenario, there are a few key roles to understand:

- **Client:** The application or agent that needs access (in our case, the AI agent).
- **Resource Server:** The API or service the agent wants to call (e.g. a company database API, MCP Server).
- **Authorization Server (IdP):** The identity provider or auth server that issues tokens and verifies identities. Often this is an enterprise IdP that supports OAuth/OIDC or dedicated STS (Secure Token Service) integrated into enterprise IDP.
- **Resource Owner:** The entity that "owns" the data or resource. For user-centric flows, this is typically the user. For machine flows, there may not be a human resource owner – the client itself is acting on its own behalf or under the organization's authority.

OAuth 2.1 defines several **grant types** (flows) by which a client can obtain an access token from the Authorization Server. The two most relevant grant types for our discussion are:

- **Client Credentials Grant:** Used for machine-to-machine authentication (no human user involved).
- **Authorization Code Grant (with PKCE):** Used for interactive user authentication and consent (human-in-the-loop scenarios).

OAuth 2.1 mandates certain security improvements: for example, **PKCE (Proof Key for Code Exchange)** is now required for all clients using the authorization code flow, to prevent interception of authorization codes. It also deprecates insecure flows like the implicit grant and password grant for better security. These changes particularly benefit scenarios like mobile apps or single-page apps, but they establish a safer default for any OAuth client – including AI agents that might use these flows.

Scopes and Consent: OAuth uses **scopes** to specify permissions and access token grants. For instance, an agent might request a token with scope **inventory:read** if it needs read-access to an inventory API. In human flows, the user would be shown a consent screen listing these scopes. In machine flows (no user present), the scopes must be pre-approved for the client by an administrator, since there is no interactive consent step. Scopes help ensure an agent's token only carries the minimum access it needs.

OpenID Connect and Identity Tokens

While OAuth 2.1 deals with authorization (letting the agent obtain access to resources), **OpenID Connect (OIDC)** deals with authentication and identity assertion—specifically, verifying and conveying user identity. OpenID Connect is built as a layer on top of OAuth 2.0/2.1. It introduces the concept of an **ID Token,** which is a token (often a JWT) that encodes identity information about a user who has authenticated.

For **human identity flows,** OIDC allows an application to know who the user is after they log in. For example, when a user logs into an application via OIDC, the app receives an ID Token containing claims about the user (such as a unique user ID, name, email, and authentication timestamp). This ID Token is intended for the **client's use only** (to establish the user's identity in the app) and is not used to access APIs.

For **AI agents**, the concept of an ID Token is a bit different, because often there is **no human user involved**. If an agent is operating autonomously (no user context), there is no human identity to assert—thus, usually **no ID Token is needed or issued** in a pure machine-to-machine scenario. The agent's "identity" in that case is its client credentials. The token it cares about is an access token proving the agent itself was authenticated by the IdP. (In some enterprise setups, an agent might still have a sort of "service identity" with attributes, but these would typically appear as claims in an access token or in a token exchange, rather than as a separate ID token).

However, if an AI agent is acting **on behalf of a user** (for instance, a digital assistant that helps a user by calling APIs in the user's account), then an ID Token would come into play because the user's identity needs to be known. In such a case, the agent would use an OIDC **authorization code flow** to log the user in, get an ID Token (identifying the user), and an access token (with the user's authorization). The agent then uses the access token to perform actions under the user's account. This scenario is essentially the same as a normal user login flow, just initiated by an agent-based application.

In summary, OpenID Connect provides:

- **ID Token:** A JSON Web Token (JWT) containing identity claims about the authenticated user (issuer, subject/user ID, audience, timestamps, etc.). It proves the user was authenticated by the IdP.
- **User Info:** (Optional) an endpoint to fetch additional profile info about the user.
- **No ID Token for Machines:** In a machine-only authentication (no user), typically no ID Token is isued because there's no human identity to represent. The focus is purely on obtaining an access token for authorization.

It's important to understand the distinction between authentication and authorization tokens as we proceed. Let's delve into the two key flows and see how tokens are obtained and used in each.

Authorization Code Flow with PKCE (Human-Interactive Flow)

The **Authorization Code Flow** (with PKCE) is the go-to OAuth 2.1 flow for applications that involve a user signing in. It's a **front-channel flow** where a user is redirected to the authorization server for login and consent, and a **back-channel exchange** where the client gets the tokens. Let's outline how this works in the context of an AI agent application that needs a user's involvement:

1. Authorization Request (User Login)
The AI agent (acting as an OAuth client) initiates the flow by redirecting the user to the Authorization Server (IdP). This request includes the client's ID, requested scopes, and a PKCE **code challenge**. PKCE stands for "Proof Key for Code Exchange," which is a mechanism to prevent attackers from stealing the authorization code. Essentially, the client first creates a random secret called the **code verifier**, then hashes it to produce the code challenge, and sends the challenge in this request.

2. User Authentication & Consent
The user is presented with a login screen by the IdP (e.g., enter username/password, use passkeys, perform MFA). After successful authentication, the IdP may show a **consent screen** listing the scopes the agent requested (e.g., "This AI assistant wants to read your inventory data"). The user consents, and then the IdP proceeds.

3. Authorization Code Issuance
After the user authenticates (and consents), the Authorization Server redirects the user-agent (browser) back to the AI agent's **redirect URI** with an **authorization code** in the URL. This code is a short-lived, one-time code.

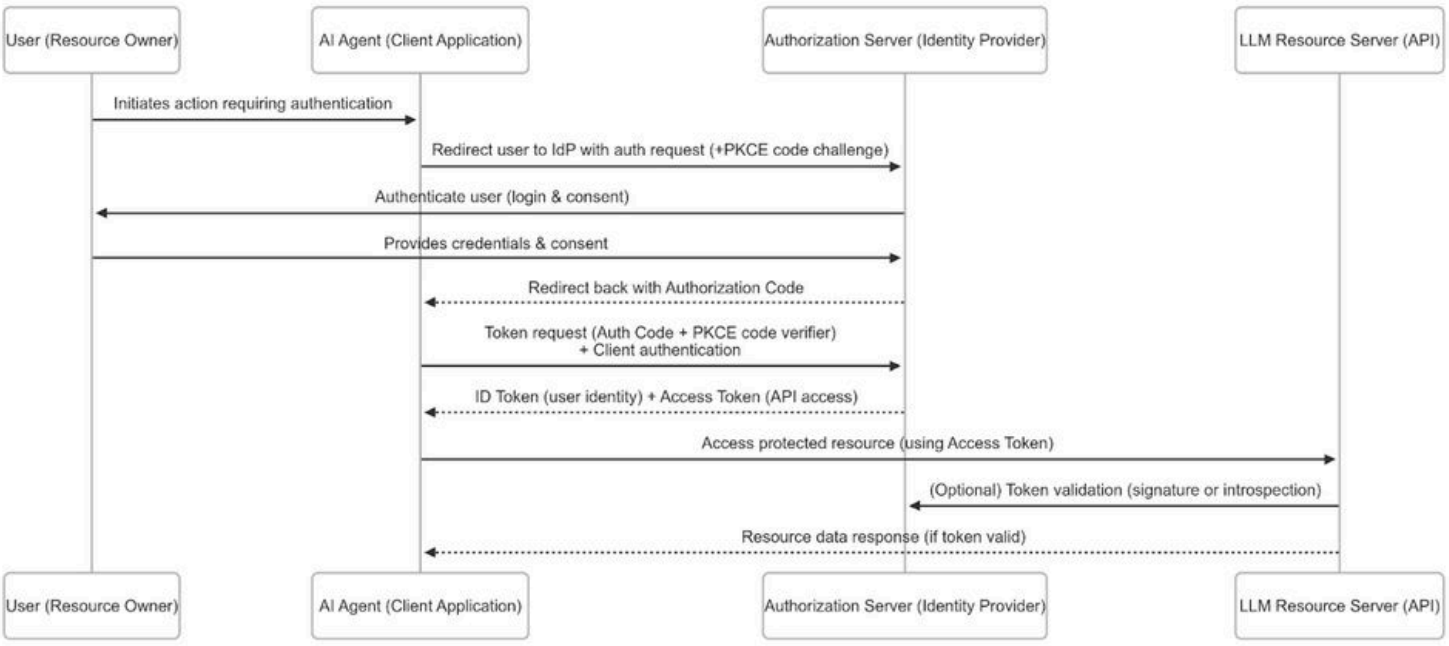
4. Code to Token Exchange (Back Channel)
The AI agent's backend now takes that authorization code and sends a **token request** to the Authorization Server's token endpoint (this is a direct server-to-server call, not through the user's browser). Along with the code, the agent sends its **client authentication** (if it's a confidential client, e.g., client ID and secret) and the **PKCE code verifier** (the original random secret). The Authorization Server verifies that the code verifier matches the earlier code challenge (proving that the entity that initiated the flow is the one redeeming the code).

5. Tokens Issued
If everything checks out, the Authorization Server responds with **tokens**—typically an **ID Token** (since this is OpenID Connect—the user's identity info) and an **Access Token** (for calling resource APIs on behalf of the user). It may also include a **Refresh Token** if the client is allowed to get one (used to get new access tokens after the current one expires, without asking the user to log in again).

6. Resource Access
The AI agent can now use the **Access Token** to call the protected **Resource Server (API)**.The API will validate the access token to ensure it's valid and intended for that API (checking **signature, expiration, audience, scopes**, etc.). If valid, the agent's request is authorized and it can perform the action (e.g., fetch inventory data). The ID Token is not sent to the API; it's only used by the agent to assert the user's identity within the agent's context if needed (for example, logging which user is associated with operations).

Authorization Code with PKCE Flow

Below is a simplified **sequence diagram** of the Authorization Code with PKCE flow, highlighting the interactions between the user, the AI agent (client), the authorization server, and the resource API:



Key points of this flow: The agent never sees the user's password; it only gets a time-limited code and then tokens. PKCE ensures the code is useless to an imposter. The presence of the **ID Token** means the client (agent) knows the identity of the user who approved the request. The **Access Token** is what the agent uses to actually get things done with the API, constrained by scopes and audience. This flow mirrors what a traditional web or mobile app would do for user login, and an AI agent application would use it in the same way when a **human user's authority or data is involved.**

Client Credentials Flow (Machine-to-Machine Flow)

When an AI agent needs to operate autonomously—for example, a script running overnight to process reports, or an AI service that coordinates data between systems—there is no interactive user present to log in. In these cases, the **Client Credentials** grant is the appropriate OAuth flow. This is a **machine-to-machine (M2M)** flow where the client (agent) authenticates directly with the Authorization Server and obtains an access token, using its own credentials. Here's how it works:

1. Direct Token Request
The AI agent (client) sends a token request directly to the Authorization Server's token endpoint, but instead of presenting an authorization code, it presents its **own credentials.** These credentials were obtained when the agent was registered in the IdP—typically a **Client ID** and **Client Secret**. In more secure setups, the agent might use a **client certificate** or a **signed JWT (private key assertion)** to authenticate itself rather than a simple static secret.

2. Authentication of Client
The Authorization Server verifies the client's credentials. Because this is a backend-to-backend call, the client must prove its identity (hence this flow is generally only used by **confidential clients** that can safely store a secret or key). Public clients (like single-page apps or mobile apps) cannot use this flow since they can't keep a secret safe; but an AI agent running on a server or secured environment can.

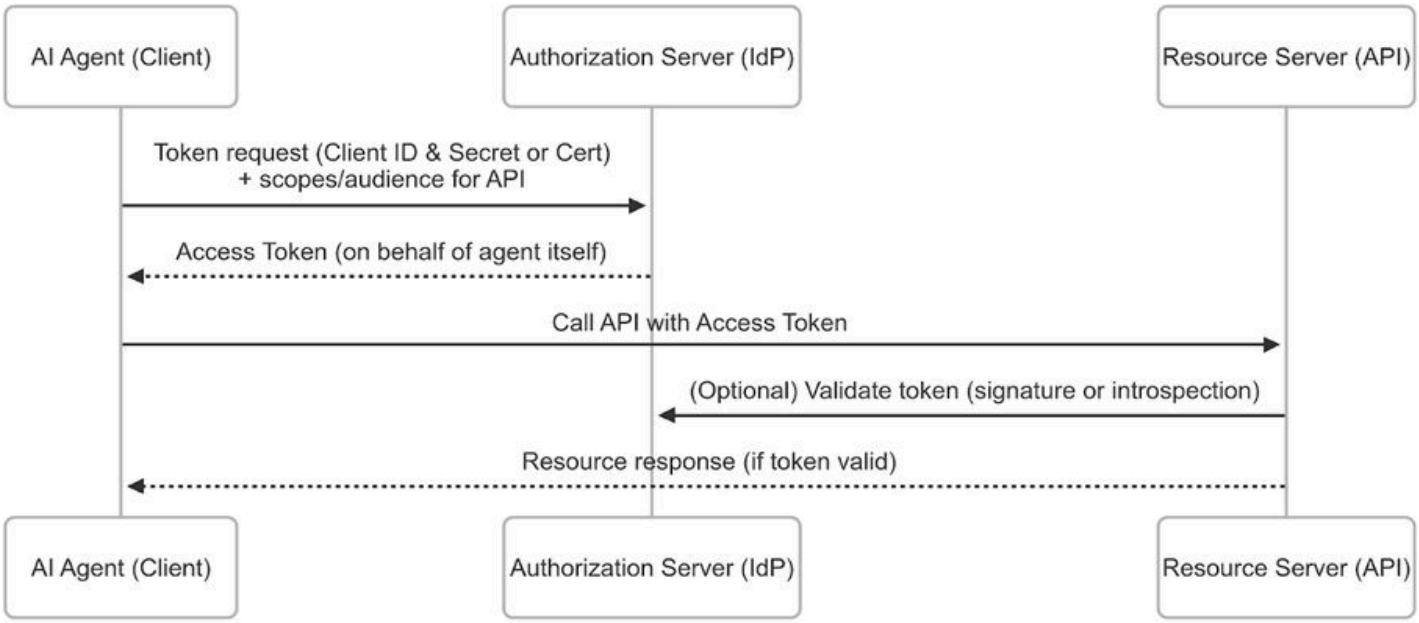
3. Scopes / Audience in Request
The agent's token request can include the desired scope of access (e.g., **reports:generate** or some permission the agent needs). In some implementations, the request might also specify an **audience** – the intended target resource for the access token—if the Authorization Server uses audience parameters. In many cases, though, the audience is implicitly determined by the client's allowed scopes or by a token request parameter.

4. Access Token Issued
If the client is authenticated and the request is valid, the Authorization Server responds with an **Access Token** (and usually **no ID Token**, since no user is involved). This access token represents the agent itself having certain permissions. The token will typically have the client's identity (or a system user identity) in its claims (e.g., in JWT, the **sub** claim could be the client ID or a service account ID) and an **aud** (audience) claim indicating which API it's meant for. It may also include scopes or other authorization details.

5. Resource Access
The AI agent then includes this access token in the Authorization header of its HTTPS requests to the **Resource Server (API)**. The Resource Server verifies the token (checking signature, expiration, and that the audience matches itself). If valid and the token grants the needed scope, the API fulfills the request. If the token is missing or invalid, the API will reject the call (usually with an HTTP 401/403 error). There is **no human** in the loop at any point.

Client Credentials Flow

The following diagram shows the Client Credentials flow sequence between the AI agent, the authorization server, and the resource API:



In this flow, since there is no user, **the Access Token is the sole token** used and it embodies the agent's permissions. There is no consent step or ID token. Security for this flow hinges on keeping the agent's credentials secure and limiting the scope of what the token allows. Typically, access tokens obtained via client credentials are scoped narrowly to specific tasks or APIs (and often have a short lifetime, like a few minutes up to an hour, to limit risk if stolen). Agents can always request a new token when one expires, using their credentials again. Because the agent is essentially "logging in" with a secret, this secret must be stored securely (e.g., in an encrypted vault or secure configuration, never hard-coded in plain text).

Access Tokens vs. ID Tokens: Purpose and Use

It's crucial to differentiate **access tokens** from **ID tokens**, as they serve different purposes in an ideneity system:

Access Tokens

This token represents **authorization**—it allows the bearer to **access certain APIs or resources**. In OAuth2/OIDC, an access token is typically a short-lived token (often a JWT or opaque value) that the client passes to the resource server. The resource server will parse or validate it to decide if the request is allowed. Access tokens contain scopes or permissions and an audience. They are meant **for the resource server**. For example, an access token might convey that "Agent X is allowed to perform **write** operations on Service Y's API, and the token is valid for 15 minutes." The resource will check that the token's audience is itself (Service Y) and that the scopes include **write**. If an access token is stolen, an attacker could potentially invoke that specific API until the token expires—which is why they are short-lived and often bound to specific audiences and scopes to minimize damage.

ID Tokens

This token represents **authentication**—it is a **proof of the identity** of a user who has logged in via OIDC. An ID token is usually a JWT signed by the IdP, containing claims about the user (subject, name, email, etc.) and information about how and when the user authenticated. Importantly, **the ID token's audience is the client** (the AI agent application, in our context)—meaning it is intended **for the client to read**. It is not supposed to be sent to APIs for authorization. For example, an ID token might tell the agent "User John Doe (**user_id 12345**) has authenticated via your IdP, and this token is issued for your client (**client_id XYZ**)." The agent can use this to personalize the experience or make access control decisions in the app, but if it tried to use the ID token to call an API, the API would reject it because the ID token's audience is not the API.

In simpler terms, **ID tokens are for identity (who), and access tokens are for access (what can be done)**. Always use the correct token for its intended purpose. An AI agent that gets both tokens from an OIDC flow will use the access token to call APIs and may use the ID token internally to record which user approved the actions or to retrieve basic user info. If the agent is operating autonomously (client credentials flow), it will only have an access token, which in a sense serves both purposes—it authenticates the agent to the API (proving the request comes from a known client) and authorizes specific actions (by scope/audience), but it doesn't represent a human identity at all.

Audience Restriction and Token Audience

One of the key security features in token-based systems is the concept of audience restriction. The "audience" (**aud**) claim in a token (particularly in JWT access tokens and ID tokens) tells you who the token is intended for. This is how we prevent a token issued for Service A from being used at Service B.

- An **ID token** typically has the audience set to the OAuth client (e.g., **aud**: **my-agent-app**). That means only **my-agent-app** should accept and process that token (which makes sense, since it's containing user info for the app). If that token were somehow sent to a resource server, the resource server would see the audience doesn't match and should refuse it.
- An **access token** will have the audience of the specific resource server or API it is meant for (or sometimes a list of audiences if it's allowed for multiple, though multiple audiences is less common in simple scenarios). For instance, if an AI agent requests an access token to call the "Inventory API", the issued token's **aud** might be **inventory-api** (some identifier for that service). When the agent calls the Inventory API, the API will check that incoming token's **aud** is **inventory-api**—if yes, proceed; if not, it might the token may have been issued for a different service, indicating potential misue or token theft, so it should be rejected.

Why is audience restriction important? Imagine an environment with multiple services (API A, API B). If an agent could use the same access token for both A and B, then a token leaked from a request to A could be replayed to B. By tying tokens to a specific audience, even if a token is compromised, it cannot be used to access a different service than intended. This containment significantly improves security in a microservices or multi-API ecosystem. Enterprises often design their OAuth scopes and audience values such that a token is narrowly scoped to one API or one logical resource server.

For AI agents, this means when the agent needs to access different APIs, it should fetch separate access tokens for each (each with the appropriate audience). In advanced scenarios, there are mechanisms like **token exchange** (which we'll explore in a later part of this series) that allow an agent to swap one token for another targeted to a different audience as it moves through an ecosystem of services. But the core principle remains: tokens should be treated as **single-resource credentials**, not universal keys.

Additionally, **scope restriction** complements audience. For example, even within the Inventory API's scope, an access token might be limited to read-only access (**inventory:read**). The API will enforce both the audience and scope—meaning the token not only must be meant for the Inventory API, but also only allows read operations if that's what was granted. This principle of least privilege is critical when designing what an AI agent can do.

Human Identity Flows vs. AI Agent Flows

Now that we've covered both major patterns, let's explicitly distinguish **human-centric identity flows** from **AI agent (machine) identity flows** in the enterprise context:

- **Interaction:** Human flows require user interaction (a person entering credentials, approving consent). AI agent flows are fully automated—no user interface, no human present to provide input. The client credentials flow is a prime example: it's a service authenticating to another service programmatically.
- **Flow Type:** Human flows use the Authorization Code grant (with PKCE in OAuth 2.1) or sometimes others like Device Code flow in special cases. AI agents typically use Client Credentials grant for standalone operation. If an AI agent needs to access user data with user consent, it temporarily switches into a "human flow" mode (auth code flow) to get permission, essentially behaving like any other app acting on behalf of a user.
- **Identity vs. Service Authentication:** Human flows establish a user's identity (hence ID tokens are issued). The agent/app is just a facilitator in that process. Machine flows establish the agent's identity (the client's own identity) to the authorization server and resource servers—no user identity is involved. The agent is both the initiator and the "owner" of the granted access in that scenario.
- **Credentials Used:** In human flows, the end-user supplies credentials (e.g., password, biometric, etc.) to the IdP. In agent flows, the agent uses its client credentials (e.g., a secret or private key) to authenticate to the IdP. The security of the flow hinges on protecting that secret key material, whereas in human flows it hinges on the user's credentials and the interactive login security (e.g., MFA).
- **Consent and Governance:** Human flows often involve user **consent** for data access (because the user might authorize the application to use their data). Enterprises may also have policy checks at this point (like asking the user to confirm they allow the AI agent app to act on their behalf). For agent flows with no user, consent is handled out-of-band—typically an administrator pre-approves what the agent can do by configuring its access. Governance for agents means managing which APIs the agent client is allowed to call, and revoking access if the agent is compromised or no longer needed. In effect, admin-controlled consent replaces end-user consent.
- **Tokens Obtained:** A human-oriented OIDC flow yields **ID token + Access token (+ possibly refresh token)**. A pure machine flow yields **Access token (often no refresh token by default)**. If long-term access is needed, either the agent can just request new tokens as needed, or in some systems a refresh token can be issued to the client as well (though often not necessary if the client can store a secret and continually re-authenticate).

To put it simply, human flows answer **"Who is the user and are they allowed?"** whereas machine flows answer **"Is this client (agent) itself allowed?"**. Both result in tokens that are used to call APIs, but the context and contents of those tokens differ.

Conclusion

In this first part of our series, we established the fundamental concepts of identity for AI agents in enterprise systems. We learned how **OAuth 2.1** provides secure mechanisms (like the Authorization Code + PKCE and Client Credentials flows) for obtaining tokens, and how **OpenID Connect** adds an identity layer with ID tokens for user authentication. We also clarified the different token types—**ID tokens vs. Access tokens**—and why **audience restrictions** and scopes are vital to ensuring tokens are used only as intended.

With this foundation, software architects can begin designing AI agent systems that integrate into the enterprise identity fabric rather than operating as unchecked black boxes. An AI agent can be thought of as a "digital employee" or service—it must authenticate and be granted only the permissions it needs, using the same robust protocols that human users and traditional services employ.

About SecureAuth

More security shouldn't mean more obstacles. Since 2005, SecureAuth has helped leading companies simplify identity and access management for customers and employees—creating experiences that are as welcoming as they are secure.

SecureAuth is redefining authentication for the modern enterprise. Today's evolving threat landscape demands innovative, adaptive security solutions. As the first-to-market provider of continuous facial authentication, we go beyond the initial authentication to deliver ongoing security throughout the entire session. Our mature AI-driven risk engine delivers dynamic—and often invisible—authentication, making you more effective than ever at eliminating threats while ensuring frictionless, secure access for employees and customers.

Welcome to Better Identity.

SECUREAUTH