Authentication & Authorization

Ashish Pratap Singh

When you log into your email, how does the system know it's really you? And once you're in, how does it know you can read your own emails but not someone else's?

The answers to these questions lie in two of the most fundamental concepts in system security: **Authentication** and **Authorization**

While they sound similar and work together, they have distinct and critical roles.

In this chapter, we will break down these two pillars of access control, explaining what they are, how they differ, and how they are implemented in modern, secure systems.

1. Introduction

In any system that involves users and data, access control is not just a feature, it's a foundational requirement. Without it, there's nothing to stop anyone from accessing sensitive information or performing destructive actions. This is where authentication and authorization come in.

Think of it like visiting a secure office building:

- **Authentication** is showing your ID badge at the front desk to prove you are an employee. The system verifies **who you are**.
- **Authorization** is what happens next. Your ID badge determines which floors and rooms you can access. The system decides **what you can do**.

Both are essential. Without authentication, anyone can walk in. Without authorization, every authenticated employee could access the CEO's office. They are distinct but interdependent processes that form the bedrock of secure system design.

2. What Is Authentication?

Authentication is the process of verifying a user's claimed identity. It answers the question, "Are you really who you say you are?" This is typically done by challenging the user to provide proof of their identity, which can come in several forms:

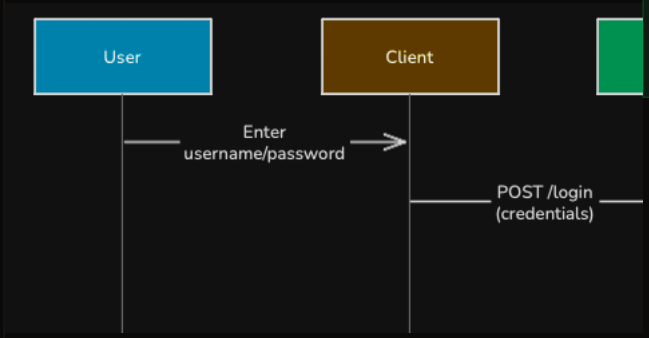
- **Something you know:** A password, PIN, or secret answer.
- **Something you have:** A physical token, a mobile phone (for OTPs), or a smart card.
- **Something you are:** A biometric factor like a fingerprint, face scan, or retina scan.

Modern systems often combine these through **Multi-Factor Authentication (MFA)** to increase security.

Common Authentication Methods

Type	Description	Example
Basic Auth	Credentials (username/password) sent with every request.	Legacy APIs
Session-Based Auth	Server issues session ID stored in cookies.	Traditional web apps
Token-Based Auth (JWT)	Stateless tokens (e.g., JWT) passed in headers.	REST APIs
OAuth 2.0	Delegated access between services.	"Sign in with Google"
Passwordless / Biometric	Email links or biometrics replace passwords.	Magic link login, Face ID

ypical Authentication Flow



xample:

User Enters Credentials: The user submits their username and password via a login form.

name and password via a login form.

2. **Server Validates:** The server receives the credentials. It finds the user in the database, **hashes** the provided password with the user's unique **salt**, and compares it to the stored hash.
3. **Server Issues Token/Session:** If the credentials are valid, the server generates a unique session ID or a signed token (like a JWT) and sends it back to the client.
4. **Client Stores Token/Session:** The client stores this identifier, typically in a secure cookie or local storage.
5. **Subsequent Requests:** For every subsequent request to the server, the client includes this token/session ID in the headers. The server uses it to quickly identify the authenticated user without needing the password again.

---

3. What Is Authorization?

Authorization is the process of determining whether an authenticated user has the necessary permissions to perform a specific action or access a particular resource. It happens *after* successful authentication and answers the question, "Is this user allowed to do this?"

Authorization decisions are made by an access policy engine based on a set of rules.

Common Authorization Models

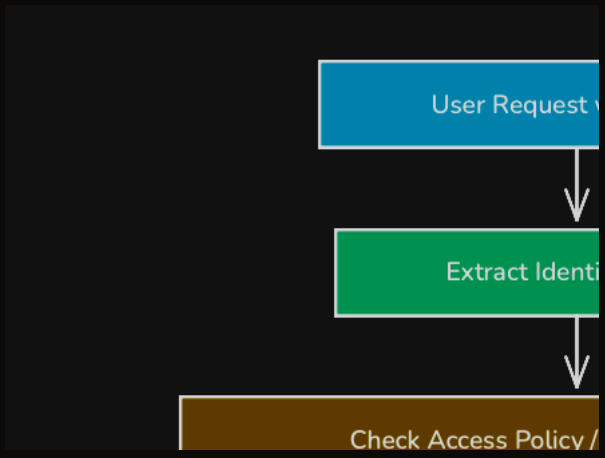
Model	Description	Example
RBAC (Role-Based Access Control)	Permissions grouped into roles.	Admin, Editor, Viewer
ABAC (Attribute-Based Access Control)	Access based on attributes like user, resource, or context.	“Allow if user.department == resource.department”
PBAC (Policy-Based Access Control)	Uses declarative policies for fine-grained control.	AWS IAM, OPA

- **Role-Based Access Control (RBAC):** Permissions are assigned to roles (e.g., `admin`, `editor`), and users are assigned to roles.
- **Attribute-Based Access Control (ABAC):** Permissions are based on attributes of the user, resource, and environment (e.g., "Allow users in the 'Marketing' depart-

ment to edit documents they own").

- **Policy-Based Access Control (PBAC):** A more flexible model where rules are defined as explicit policies.

Typical Authorization Flow



Example:

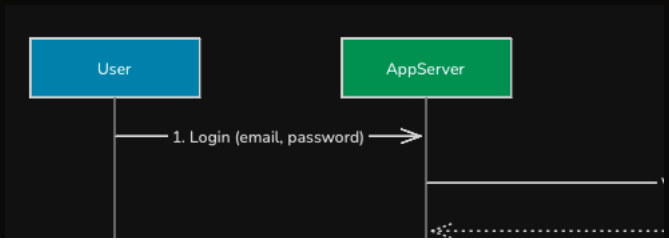
1. **User Makes a Request with Token:** The user, now logged in, tries to perform an action, like `DELETE /api/posts/123`. The request includes their authentication token.
2. **Server Extracts Identity:** The server validates the token and extracts the user's identity and associated roles (e.g., `userId: 5`, `roles: ['editor']`).
3. **Authorization Logic Checks Permissions:** The server's authorization logic checks its access control policy. Does the `editor` role have the `delete:post` permission?
4. **Access Granted or Denied:** Based on the policy, the server either executes the action and returns a `200 OK` response or denies it and returns a `403 Forbidden` error.

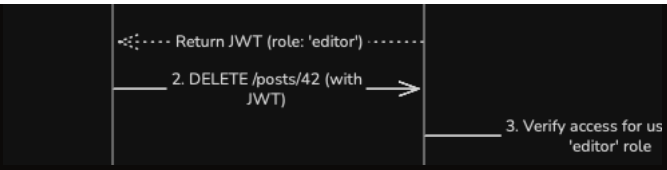
---

4. Authentication & Authorization in Action

Let's see how it all comes together.

Scenario: An editor logs into a blog platform and tries to delete a post.





1. **Authentication:** The editor enters their email and password. The server verifies them and returns a JWT containing `userId: 10` , `role: 'editor'` .
2. **Authorization Request:** The editor clicks "Delete" on a post. The client sends a `DELETE /api/posts/42` request with the JWT in the `Authorization` header.
3. **Authorization Check:**
    - The server validates the JWT's signature.
    - It extracts the `role: 'editor'` claim.
    - The authorization logic checks if the `editor` role has the `delete:post` permission.
4. **Decision:** The policy allows it, so the server deletes the post from the database and returns a `204 No Content` response. If the user had a `viewer` role, the server would have returned a `403 Forbidden` error.

---

5. Security Considerations & Best Practices

- **Always use HTTPS (TLS):** Encrypt all communication to prevent credentials and tokens from being intercepted.
- **Never store plain passwords:** Always hash passwords with a strong, slow algorithm (like Argon2 or bcrypt) and a unique salt for each user.
- **Use short-lived tokens:** Access tokens should have a short expiry time (e.g., 15 minutes), with a long-lived refresh token used to get new ones.
- **Implement the Principle of Least Privilege:** Users should only have the minimum permissions necessary to do their job.
- **Guard against common attacks:** Sanitize inputs, use anti-CSRF tokens for session-based apps, and implement rate limiting on login endpoints.