Web Architecture & Real-Time Engineering Notes

Comprehensive Technical Guide on JWT, OAuth 2.0 / OIDC, and WebSockets

---

1. Authentication & Authorization via JWT (JSON Web Tokens)

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. It is most commonly used for stateless authentication and fine-grained authorization in modern microservices and web applications.

1.1 Anatomy of a JWT Structure

A JWT string consists of three distinct parts separated by dots (.), formatted as:
Header.Payload.Signature

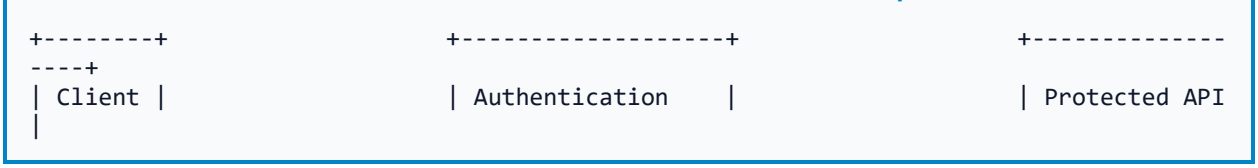
- **Header:** Contains metadata about the token, typically specifying the token type ('JWT') and the signing algorithm used (e.g., HS256 for HMAC SHA-256 or RS256 for RSA/Asymmetric).
- **Payload:** Contains the claims (statements about an entity, usually the user, along with additional metadata). Standard claims include 'sub' (subject), 'iss' (issuer), 'exp' (expiration time), 'iat' (issued at), and custom claims such as user roles.
- **Signature:** Generated by taking the Base64Url-encoded header, Base64Url-encoded payload, and signing it with a secret key (symmetric) or a private key (asymmetric). It guarantees token integrity.

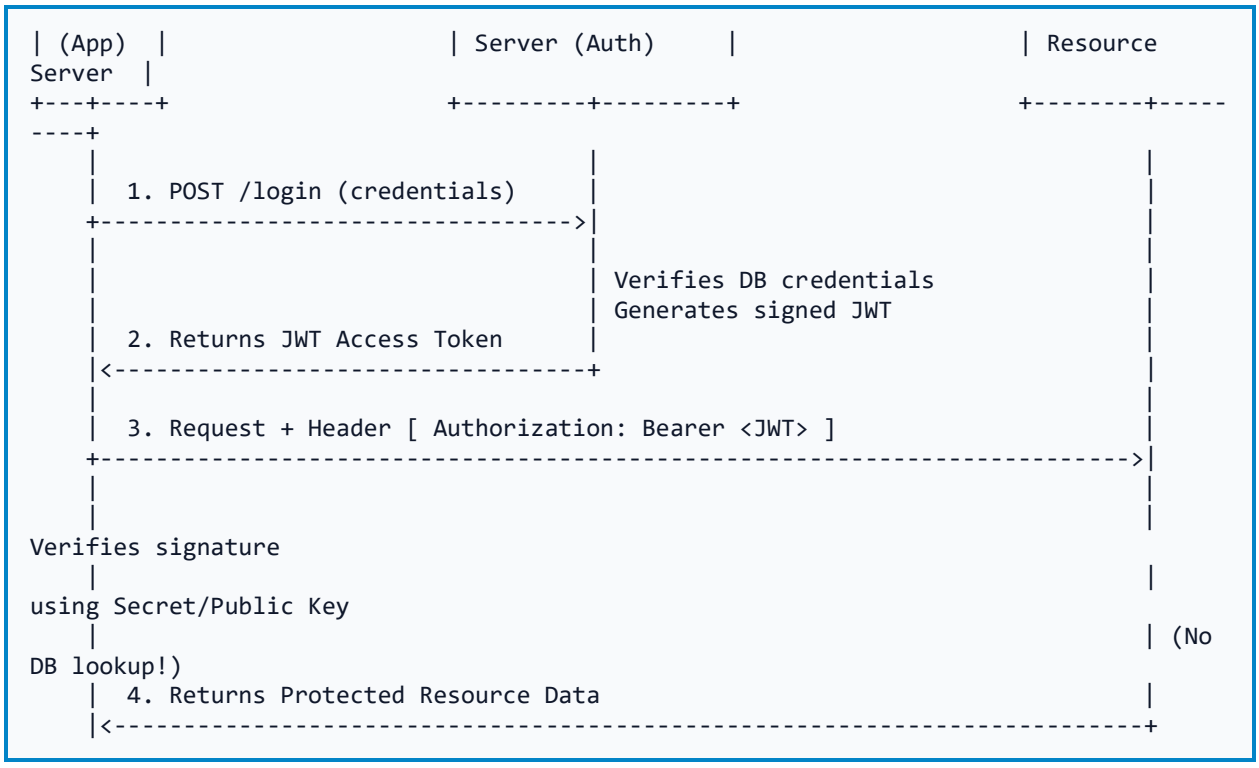
```
// Example Base64Url Encoded JWT:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkFsaWNlIiwicm9sZSI6ImFkbWluIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

1.2 JWT Authentication Architecture Flow Diagram

The diagram below illustrates how a client authenticates with an identity server, receives a JWT, and uses it to access protected backend REST APIs statelessly without requiring server-side session lookup.

FIGURE DIAGRAM: JWT Authentication Flow Sequence





1.3 JWT Security Best Practices

- **Storage Security:** Store access tokens in memory or HttpOnly, SameSite, Secure cookies to mitigate Cross-Site Scripting (XSS). Avoid storing raw tokens in LocalStorage.
- **Token Lifespan:** Keep JWT expiration times short (e.g., 5 to 15 minutes) and issue long-lived Refresh Tokens (stored safely in HttpOnly cookies) to obtain fresh access tokens.
- **Algorithm Integrity:** Always use strong algorithms (RS256 or ES256 for public-key cryptography; HS256 with long 256-bit secrets for symmetric). Explicitly reject the 'none' algorithm.

2. Delegated Authorization & Identity using OAuth 2.0 / OIDC

OAuth 2.0 is an industry-standard protocol for authorization that allows third-party applications to grant limited access to user resources without exposing user credentials. OpenID Connect (OIDC) is an identity layer built on top of OAuth 2.0 that provides user authentication and identity verification (ID Tokens).

2.1 Key Terminology & Roles

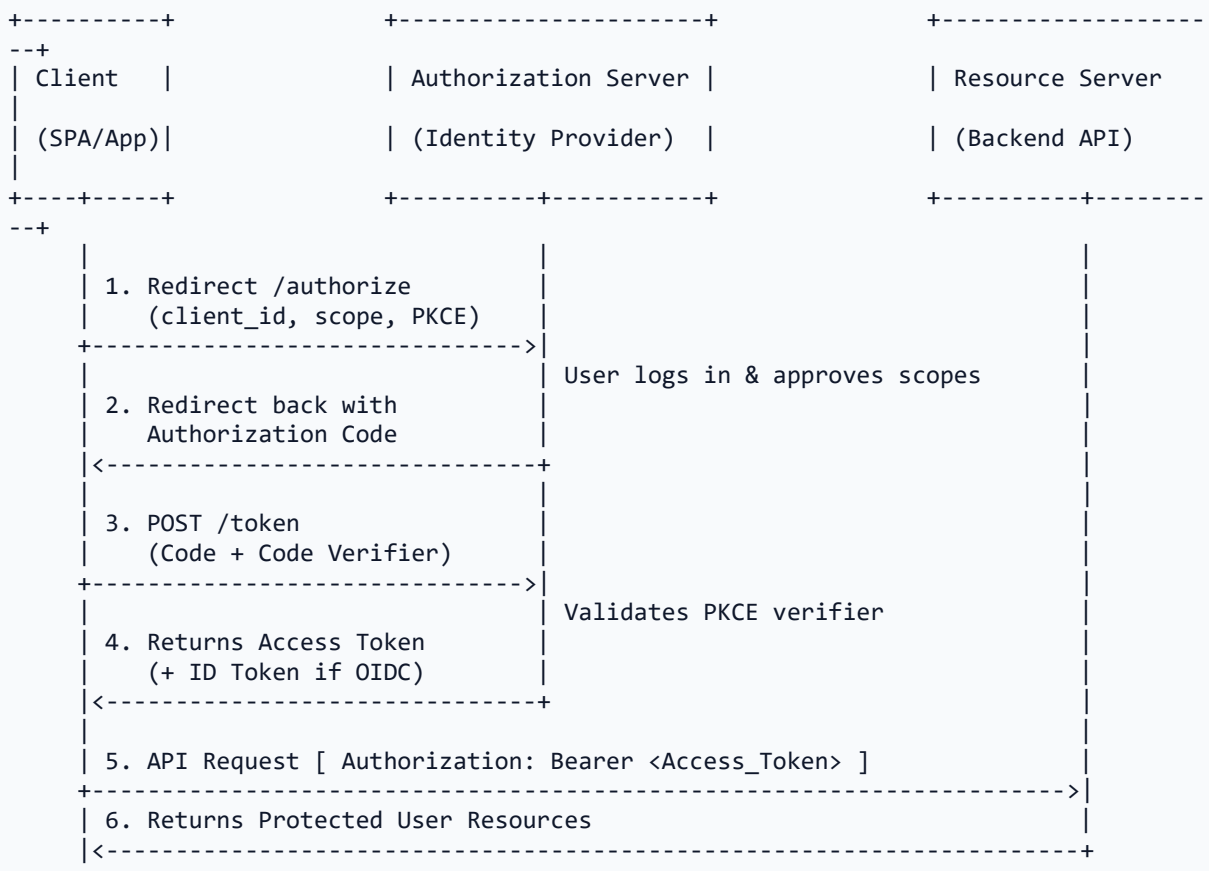
- **Resource Owner:** The end-user who owns the data and grants access to their account.
- **Client:** The application requesting access to the resource owner's account (e.g., Web App, Mobile App).

- **Authorization Server:** The server issuing access tokens after successfully authenticating the user (e.g., Auth0, Keycloak, Okta, Google).
- **Resource Server:** The API server hosting the protected resources that accepts and validates access tokens.

2.2 OAuth 2.0 Authorization Code Grant with PKCE Diagram

The Authorization Code Flow with PKCE (Proof Key for Code Exchange) is the modern gold standard security flow for single-page apps (SPAs), mobile apps, and server-side web applications.

FIGURE DIAGRAM: OAuth 2.0 Authorization Code Flow with PKCE



2.3 Comparison: JWT vs. OAuth 2.0 vs. OpenID Connect

Concept	Primary Purpose	Artifact / Protocol Type	Key Characteristic
JWT	Data format / Token container	Format Standard (RFC 7519)	Self-contained, cryptographically signed payload

OAuth 2.0	Delegated Authorization	Protocol / Framework (RFC 6749)	Grants scoped access to APIs via Access Tokens
OpenID Connect	User Authentication & ID	Identity Layer on OAuth 2.0	Provides profile info via standardized ID Token

3. Real-Time Communication using WebSockets

WebSocket is a computer communications protocol (RFC 6455) providing full-duplex, bi-directional communication channels over a single persistent TCP connection. Unlike HTTP request-response architecture, WebSockets enable servers to push data to clients instantly without polling.

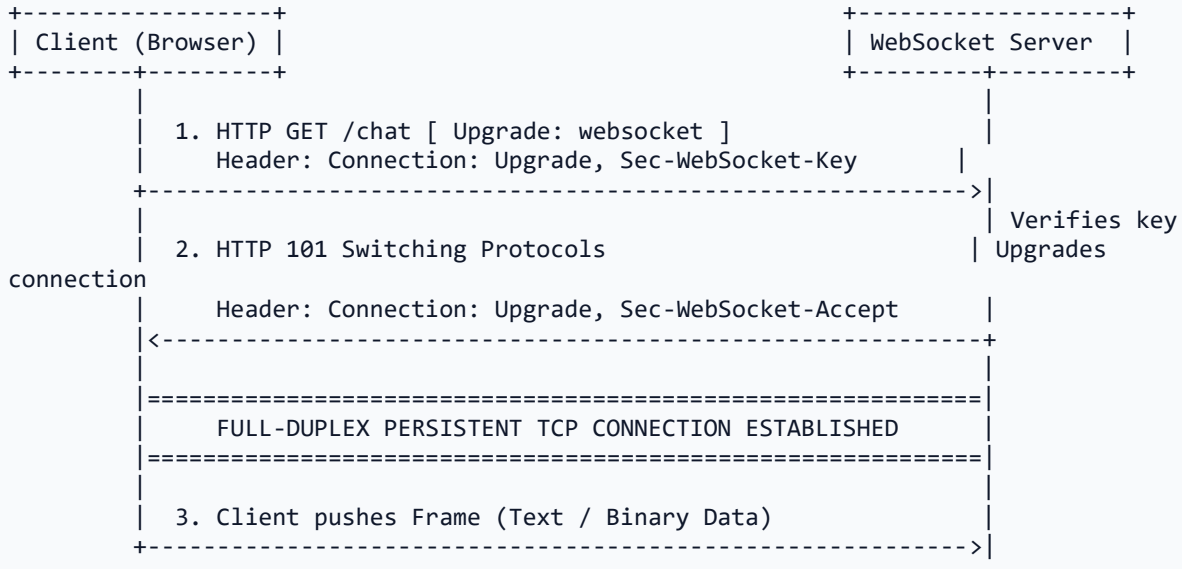
3.1 HTTP vs. WebSockets Comparison

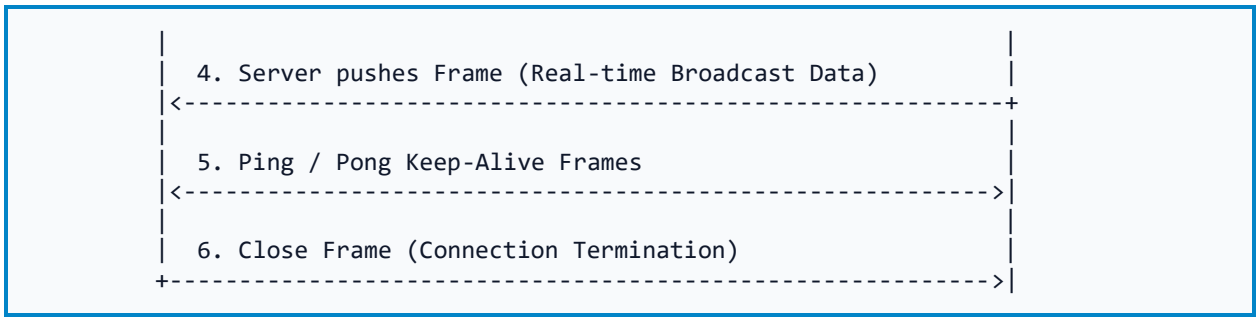
- **Traditional HTTP:** Half-duplex request-response model. High header overhead (headers sent with every request). Unsuitable for low-latency real-time updates.
- **WebSockets (ws:// or wss://):** Full-duplex bi-directional connection. Minimal frame overhead (2 to 10 bytes per frame after initial handshake). Ideal for chat, live sports, stock tickers, and collaborative editing.

3.2 WebSocket Handshake & Life Cycle Diagram

A WebSocket connection starts as an HTTP/1.1 request with an 'Upgrade' header. Once the server accepts the upgrade, the underlying TCP socket remains open for bi-directional frame transfer.

FIGURE DIAGRAM: WebSocket Protocol Handshake & Communication Lifecycle





3.3 Combining JWT Authentication with WebSockets

Because standard WebSocket connections initiated from browser JS (`new WebSocket(url)`) do not allow custom HTTP request headers, passing JWTs requires specific strategies:

- **Query Parameter Token:** Pass the JWT as a ticket or query parameter during handshake (e.g., `wss://api.example.com/socket?token=<JWT>`). Note: Query params can be logged in proxy servers.
- **First-Message Authentication:** Establish WebSocket connection, then immediately send an initial auth JSON payload (`{ type: 'AUTH', token: '<JWT>' }`). If unauthenticated within a timeout (e.g., 5s), the server terminates the connection.
- **Cookie-Based Handshake:** Rely on HttpOnly cookies sent automatically during the initial HTTP 101 handshake request.

```
// Example Node.js/JavaScript Client First-Message Auth:
const socket = new WebSocket('wss://api.example.com/ws');

socket.onopen = () => {
    // Send JWT token immediately upon connection
    socket.send(JSON.stringify({
        type: 'AUTHENTICATE',
        token: userJwtToken
    }));
};

socket.onmessage = (event) => {
    const message = JSON.parse(event.data);
    console.log('Real-time data received:', message);
};
```