JSON Web Tokens (JWT)

Version: 1.0
Last updated: December 11, 2025

Table of contents

---

1. What is JWT?

JSON Web Token (JWT) is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object. The token is compact, URL-safe, and can be digitally signed (JWS) and/or encrypted (JWE).

JWTs are commonly used for stateless authentication and authorization in web and mobile applications.

2. JWT structure

A compact serialized JWT has three base64url-encoded parts separated by dots:

```
HEADER.PAYLOAD.SIGNATURE
```

Header

Includes metadata such as the signing algorithm and token type. Example:

```
{ "alg": "HS256", "typ": "JWT" }
```

Payload (Claims)

Holds the statements about an entity (user) and additional data. Claims can be registered (standard), public, or private (custom).

Example payload:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "roles": ["user","admin"],
  "iat": 1516239022,
  "exp": 1516242622
}
```

Signature

The signature ensures token integrity. For HMAC using SHA-256:

```
HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret )
```

3. Example JWT (compact)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpv
aG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

(Decoded, header/payload shown earlier.)

4. Common use-cases

- Authentication (stateless session tokens)
- Authorization (carry user roles / scopes)
- Information exchange (signed claims between services)
- Single Sign-On (SSO) via OpenID Connect

5. Signing (JWS) & Encryption (JWE)

- **JWS** — JSON Web Signature: signs/verifies integrity. Algorithms include HS256, RS256, ES256, PS256.
- **JWE** — JSON Web Encryption: encrypts content for confidentiality.

Common recommendation: use **RS256** (asymmetric keys) or **ES256** for stronger security in production; HS256 (shared secret) is simpler but requires careful secret management.

6. Standard (registered) claims

- **iss** — issuer
- **sub** — subject (usually user id)
- **aud** — audience
- **exp** — expiration time (epoch seconds)
- **nbf** — not before
- **iat** — issued at
- **jti** — JWT ID (unique identifier)

Use `exp` and `iat` always for access tokens.

7. Custom claims & best practices

- Keep payload small — tokens are sent on each request.
- Don't store secrets or sensitive data (passwords, PII) in payload unless encrypted (JWE).
- Use namespaced custom claims (e.g., `https://example.com/roles`) to avoid collisions with registered claims.

8. Token lifecycle

Issuance

Server creates a JWT after authenticating a client and returns it to the client.

Validation

- Verify signature using appropriate key.
- Check `exp`, `nbf`, and optionally `iat`.
- Verify `iss` and `aud` if used.

Expiration & rotation

- Keep access tokens short-lived (minutes to hours).
- Use refresh tokens (longer-lived, rotate on use) to obtain new access tokens.

Revocation

JWTs are stateless and can't be invalidated server-side without extra mechanisms. Common approaches: - Token blacklist (store revoked `jti` in server cache/DB). - Maintain short `exp` times with refresh tokens. - Use a token version or `token_issued_at` stored in user DB — reject tokens with `iat` earlier than the stored value.

9. Storage & transport

Storage options

- **Cookies (HttpOnly, Secure, SameSite)** — safer against XSS if marked HttpOnly and Secure; CSRF must be mitigated (SameSite or CSRF tokens).
- **LocalStorage / SessionStorage** — simple but vulnerable to XSS.
- **In-memory (SPA)** — safer than persistent storage but lost on tab close.

Transport

- Always send over TLS (HTTPS).
- Prefer Authorization header: `Authorization: Bearer <token>`.

10. Refresh tokens vs access tokens

- **Access token**: short-lived, used for API calls.
- **Refresh token**: long-lived, used to get new access tokens. Keep refresh token storage secure (HttpOnly cookie recommended) and implement rotation (issue a new refresh token on each use and revoke the old one).

11. Security considerations & common vulnerabilities

- **XSS**: Avoid storing tokens in JavaScript-accessible storage.
- **CSRF**: If storing JWT in cookies, use `SameSite=strict/lax` and CSRF tokens for state-changing requests.
- **Token leakage**: Log carefully; never log tokens.
- **Algorithm confusion attacks**: Do not accept `alg: none` and prefer explicit `alg` checks; prefer asymmetric algorithms (RS256/ES256) to avoid shared-secret key misuse.
- **Replay attacks**: Use `jti`, short `exp`, and refresh token rotation/one-time-use refresh tokens.
- **Weak keys**: Ensure secrets and private keys are sufficiently long and protected; use managed secrets (vaults) in production.

12. Implementations & code snippets

Java — Spring Boot (using `io.jsonwebtoken:jjwt` or `auth0/java-jwt`)

```
// Using jjwt (simplified)
String jwt = Jwts.builder()
    .setSubject(userId)
    .claim("roles", roles)
    .setIssuedAt(new Date())
    .setExpiration(Date.from(Instant.now().plus(15, ChronoUnit.MINUTES)))
    .signWith(secretKey, SignatureAlgorithm.HS256)
    .compact();

// Validate
Jws<Claims> parsed = Jwts.parserBuilder()
    .setSigningKey(secretKey)
    .build()
    .parseClaimsJws(jwt);
```

Node.js — jsonwebtoken

```
const jwt = require('jsonwebtoken');
const token = jwt.sign({ sub: user.id, roles: user.roles },
process.env.JWT_SECRET, { expiresIn: '15m' });

// Verify
const payload = jwt.verify(token, process.env.JWT_SECRET);
```

Python — PyJWT

```
import jwt
token = jwt.encode({'sub': user_id, 'exp': datetime.utcnow() +
timedelta(minutes=15)}, SECRET, algorithm='HS256')
payload = jwt.decode(token, SECRET, algorithms=['HS256'])
```

.NET Core

```
var tokenHandler = new JwtSecurityTokenHandler();
var tokenDescriptor = new SecurityTokenDescriptor
{ /* subject, expires, signing creds */ };
var token = tokenHandler.CreateToken(tokenDescriptor);
var jwt = tokenHandler.WriteToken(token);
```

13. Integration patterns: OAuth2 / OpenID Connect

- JWTs are used as access tokens or ID tokens in OAuth2 and OpenID Connect flows.
- When using OIDC, ID Token contains user identity claims and is intended for the client; access token is for resource servers.
- Use provider metadata (`.well-known/openid-configuration`) to fetch keys (JWKS) and validate `kid` header when using RS256.

14. Debugging & tooling

- Use jwt.io debugger for quick decoding and verifying signature (do not paste production secrets there).
- Log validation errors with caution (avoid logging tokens directly).
- Use JWT introspection endpoints (OAuth2) if using opaque tokens or provider-based token checks.

15. Performance & scalability

- JWT verification (esp. asymmetric) involves crypto; cache public keys (JWKS) with TTL.
- Keep tokens small to reduce network overhead (avoid large claims arrays).
- For token revocation lists, use fast in-memory stores (Redis) with TTL aligned to token expiry.

16. Checklist & best practices summary

- Always use HTTPS.
- Pick the right algorithm (RS/ES over HS for distributed systems).
- Keep short lifetime for access tokens; use refresh tokens.
- Store tokens securely (HttpOnly cookie preferred for web apps).
- Implement refresh token rotation and server-side revocation strategy.
- Validate `iss`, `aud`, `exp`, `nbf`, and signature.
- Don’t store sensitive PII in payload unless encrypted (JWE).
- Rotate signing keys when needed and support key identifiers (`kid`) in header.

Appendix: Quick reference

- RFCs: RFC 7519 (JWT), RFC 7515 (JWS), RFC 7516 (JWE), RFC 7517 (JWK), RFC 7518 (algorithms)
- Useful libraries: `jjwt`, `auth0/java-jwt`, `jsonwebtoken` (Node), `PyJWT`, `System.IdentityModel.Tokens.Jwt` (.NET)

If you want, I can: - Add a full worked example project (Spring Boot) with token issuance, filter-based validation, refresh token rotation, and a small front-end sample. - Produce short cheat-sheets for developers or architects (1-page summary). - Convert this into a printable PDF or slide deck.

Tell me which of the above you'd like next and I will prepare it.