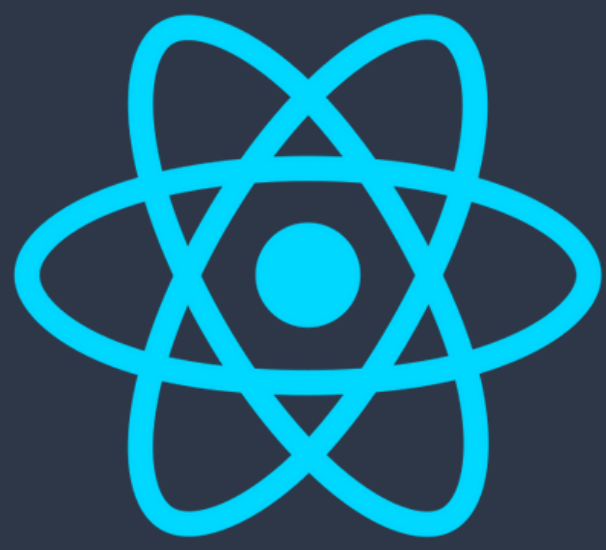
Shailesh Diwanji

JWT Token

Complete guide for JSON Web Token




📋 Table of Contents

1. What is JWT?

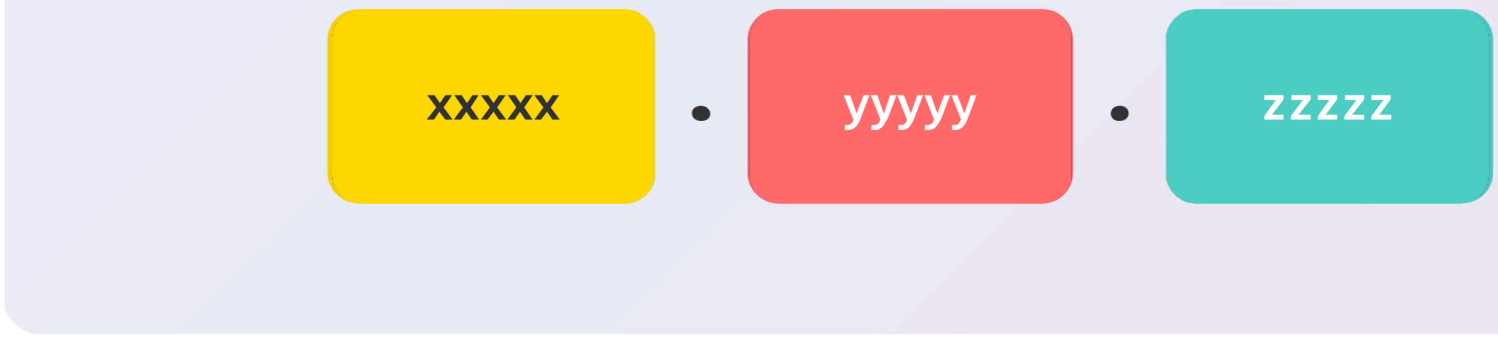
JWT (JSON Web Token) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

> **Key Point:** JWTs can be signed using a secret (with HMAC algorithm) or a public/private key pair (using RSA or ECDSA).

JSON Web Tokens are widely used in modern web applications because they provide a stateless authentication mechanism. Unlike traditional session-based authentication, JWT doesn't require the server to store session information, making it ideal for distributed systems and microservices architectures.

2. JWT Structure

A JWT consists of three parts separated by dots (.), which are:



2.1 Header

The header typically consists of two parts: the type of the token (JWT) and the signing algorithm being used, such as HMAC SHA256 or RSA.

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

2.2 Payload

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims:

- **Registered claims:** Predefined claims like `iss` (issuer), `exp` (expiration time), `sub` (subject)

- **Public claims:** Custom claims created to share information between parties that use JWTs
- **Private claims:** Custom claims created to share information between parties that agree on using them

```
{ "sub": "1234567890", "name": "John Doe", "email":
"john@example.com", "admin": true, "iat":


1516239022, "exp": 1516242622 }
```

2.3 Signature

The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way. To create the signature part, you have to take the encoded header, the encoded payload, a secret, and the algorithm specified in the header.

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

3. How JWT Authentication Works

The JWT authentication flow follows these steps:

Step 1: User Login

The user submits their credentials (username and password) to the authentication server through a login form or API endpoint.

Step 2: Server Validation

The server validates the credentials against the database. If the credentials are correct, the process continues; otherwise, an error is returned.

Step 3: JWT Generation

If the credentials are valid, the server generates a JWT containing user information (like user ID, email, roles) and signs it with a secret key.

Step 4: Token Response

The server sends the JWT back to the client, typically in the response body of the login request.

Step 4: Token Response

The server sends the JWT back to the client, typically in the response body of the login request.

Step 5: Token Storage

The client stores the JWT, usually in localStorage, sessionStorage, or as an HTTP-only cookie.

Step 6: Authenticated Requests

For subsequent requests to protected resources, the client includes the JWT in the Authorization header using the Bearer schema:

```
Authorization: Bearer <token>
```

Step 7: Token Verification

The server receives the request, extracts the JWT from the Authorization header, and verifies its signature using the secret key. If valid, it decodes the payload to get user information.

Step 8: Access Granted

If the token is valid and not expired, the server processes the request and returns the appropriate response. If invalid or expired, it returns an authentication error.

4. Advantages of JWT

🚀 Stateless

No need to store session information on the server, reducing database queries and memory usage.

📈 Scalability

Ideal for distributed systems and microservices architecture. Servers don't need to share session data.

🌐 Cross-Domain

Works seamlessly across different domains and platforms. Perfect for APIs consumed by multiple clients.

📱 Mobile-Friendly

Perfect for mobile applications that communicate with backend APIs. No cookie management required.

⚡ Performance

Reduces server load as no session lookup is required. All necessary information is in the token.

🔄 Decoupling

Authentication server can be completely separate from application servers, improving architecture flexibility.

5. Security Best Practices

- **Use HTTPS Always:** Always transmit JWTs over secure HTTPS connections to prevent token interception by attackers.
- **Set Expiration Time:** Always include an expiration time (`exp` claim) in your tokens. Short-lived tokens (15-30 minutes) are more secure.
- **Keep Secrets Secret:** Store signing keys securely using environment variables or secret management systems. Never expose them in client-side code.
- **Use Strong Algorithms:** Prefer RS256 (RSA Signature with SHA-256) over HS256 for production applications, especially for public APIs.
- **Validate Everything:** Always validate the token signature, expiration time, and claims on the server side. Never trust the client.
- **Don't Store Sensitive Data:** Remember that JWTs are encoded (not encrypted). Anyone can decode them. Don't include passwords, credit card numbers, or other sensitive information.
- **Implement Token Refresh:** Use short-lived access tokens combined with longer-lived refresh tokens for better security.

- **Use CSRF Protection:** When storing JWTs in cookies, implement CSRF (Cross-Site Request Forgery) protection mechanisms.
- **Implement Token Blacklisting:** For critical applications, maintain a blacklist of revoked tokens in a fast cache (like Redis).
- **Monitor and Log:** Log authentication attempts and monitor for suspicious patterns like rapid token generation or unusual access patterns.

6. Common Use Cases

- **Authorization:** This is the most common scenario for using JWT. Once a user is logged in, each subsequent request includes the JWT, allowing the user to access routes, services, and resources that are permitted with that token.

  Information Exchange: JWTs are a good way of securely transmitting information between parties because they can be signed, which means you can be sure the senders are who they say they are.

- **Single Sign-On (SSO):** JWT is widely used for SSO because of its small overhead and ability to be easily used across different domains. Users can access multiple applications with one set of login credentials.

- **API Authentication:** RESTful APIs commonly use JWT for authentication. The stateless nature of JWT makes it perfect for API authentication where maintaining server-side sessions would be impractical.

- **Mobile Applications:** Mobile apps use JWTs to maintain user sessions without complex cookie management, making the authentication process smoother and more reliable.

- **Microservices Communication:** In microservices architectures, JWTs can be passed between services to maintain user context without requiring a centralized session store.

7. JWT vs Session Authentication

Feature	JWT	Session
Storage Location	Client-side (localStorage, cookies)	Server-side (database, memory)
Scalability	Highly scalable (stateless)	Less scalable (stateful)
Server Load	Lower (no database lookup needed)	Higher (requires session lookup)
Revocation	Difficult (requires blacklisting)	Easy (delete from server)
Token/Session Size	Larger (contains user data)	Smaller (just session ID)
Cross-Domain Support	Excellent (works everywhere)	Limited (cookie domain restrictions)
Security	Secure if implemented correctly	Secure but requires secure storage
Implementation Complexity	Moderate (token management)	Simple (built into frameworks)

8. Code Examples

Creating a JWT (Node.js with jsonwebtoken)

```
const jwt = require('jsonwebtoken'); // Create a
token const payload = { userId: 123, email:
'user@example.com', role: 'admin' }; const secret =
process.env.JWT_SECRET; const options = { expiresIn:
'1h', // Token expires in 1 hour issuer: 'myapp.com'
}; const token = jwt.sign(payload, secret, options);
console.log('Generated Token:', token);
```

Verifying a JWT (Node.js)

```
const jwt = require('jsonwebtoken'); const token =
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'; const
secret = process.env.JWT_SECRET; try { const decoded
= jwt.verify(token, secret); console.log('Token is
valid. User ID:', decoded.userId); console.log('User
email:', decoded.email); } catch (error) { if
(error.name === 'TokenExpiredError') {
console.log('Token has expired'); } else if
(error.name === 'JsonWebTokenError') {
console.log('Invalid token'); } else {
console.log('Token verification failed:',
error.message); } }
```

Conclusion

JWT (JSON Web Token) has become the de facto standard for authentication in modern web applications, particularly for RESTful APIs and Single Page Applications (SPAs). Its stateless nature, scalability, and flexibility make it an excellent choice for distributed systems and microservices architectures.

While JWT offers numerous advantages, it's crucial to implement it correctly with proper security measures. Understanding when to use JWT versus traditional session-based authentication is important for making informed architectural decisions.

As web applications continue to evolve, JWT remains a fundamental technology that every developer should understand and know how to implement securely.