JWT (JSON Web Token)

What is JWT?

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. This information can be verified and trusted because it is digitally signed using a secret (HMAC algorithm) or a public/private key pair (RSA or ECDSA). JWTs are commonly used for **authentication** and **authorization** in web applications, APIs, and microservices architectures.

JWTs are commonly used for authentication and authorization in web applications, APIs, and microservices architectures.

Why is JWT Used?

JWT is widely used due to its efficiency and security features. Some of its key benefits include:

1. **Stateless Authentication**: JWT eliminates the need for a session state stored on the server, allowing scalability in distributed systems.
2. **Compact & Fast**: Being a compact token, JWTs are easily transmitted over HTTP headers, making them suitable for web applications and mobile devices.
3. **Security**: JWTs can be signed (integrity protection) and optionally encrypted (confidentiality protection), ensuring secure data transmission.
4. **Interoperability**: JWTs are widely adopted across different programming languages and frameworks.
5. **Fine-grained Access Control**: JWT can include claims (permissions, user roles) in the payload, allowing precise access control.

Why JWT was introduced: Drawbacks of Basic Authentication

JWTs were introduced to address the limitations of older authentication methods like **Basic Authentication**. While Basic Authentication is still widely used for simple cases, it has several **drawbacks** that JWT helps to overcome:

1. Security of Credentials:

- **Basic Authentication** sends the username and password with every HTTP request, which increases the risk of exposing sensitive credentials, especially over unencrypted channels. Even though Basic Auth often uses HTTPS, frequent transmission of usernames and passwords can still be vulnerable to man-in-the-middle (MITM) attacks if not handled properly.
- **JWT** addresses this by transmitting a token that is signed and can be verified, meaning credentials like passwords are not repeatedly sent across requests. Instead, a token is issued once upon successful authentication and then used for further requests, reducing exposure of the password.

2. Statelessness & Scalability:

- In **Basic Authentication**, the server needs to authenticate the user for each request, often requiring session management (storing user states on the server). This can become a bottleneck, especially in large, distributed applications.

- **JWT** enables **stateless** authentication. Once the server issues the JWT, the client stores it (usually in memory or local storage) and sends it with every request. The server doesn’t need to store session information, making JWT-based authentication highly scalable. The server only needs to verify the token on each request, making it more efficient.

3. Session Management:

- **Basic Authentication** relies on session-based authentication, meaning each session requires the server to maintain state information (such as user identity or session expiration) between requests. This can become cumbersome, especially for distributed systems or when users need to authenticate across different platforms.
- **JWT** eliminates the need for server-side sessions, as the token itself contains all the necessary claims (e.g., user ID, expiration time, permissions). The token is self-contained and portable, so it can be easily used across different servers, APIs, and services. The server just needs to verify the token's signature and check its expiration.

4. Cross-Domain Authentication (Single Sign-On):

- **Basic Authentication** does not work well in environments where a user must authenticate across multiple domains or services (e.g., Single Sign-On, or SSO). Each service would need to authenticate users individually, which can be inefficient and cumbersome.
- **JWT** is particularly useful for **Single Sign-On (SSO)**, where the same token can be used across different domains or systems. Once the user is authenticated, they can use the JWT to authenticate to multiple systems without needing to log in again.

5. Token Expiration & Revocation:

- **Basic Authentication** does not provide built-in mechanisms for expiring or revoking authentication tokens. If a session is hijacked, it can continue until the user manually logs out or the session expires (which might be controlled via a timeout, but that can still be insecure).
- **JWT** tokens can have **built-in expiration times** (via the `exp` claim), after which they become invalid. Additionally, JWT-based systems can implement **refresh tokens** to issue new JWT tokens when the old one expires. If a JWT is compromised, it can also be revoked by invalidating the refresh token or by checking a blacklist.

6. Flexibility and Rich Claims:

- **Basic Authentication** is limited to transmitting only the username and password. If you need more information (e.g., user roles, permissions, or expiration time), it requires extra steps like querying the database after authentication.
- **JWT** allows you to include rich information (or "claims") in the token itself. Claims can include user roles, permissions, token expiration, and other attributes, making JWT highly customizable. This reduces the need for additional queries or complex database lookups.

How JWT Works:

JWT tokens typically consist of three parts:

1. **Header**: Contains the metadata about the token (e.g., the signing algorithm used, such as HMAC or RSA).
2. **Payload**: Contains the claims or data that you want to transmit (e.g., user ID, permissions, expiration time).
3. **Signature**: A cryptographic signature to verify that the token has not been tampered with. This signature is generated using a secret key or a public/private key pair.

When a user logs in, the server generates a JWT token with the user's information and signs it. The user then includes this token in subsequent requests, and the server verifies the token's validity (using the signature) before granting access to resources.

While Basic Authentication is simple and effective for small or internal applications, JWT provides a more scalable, secure, and flexible solution for modern web applications, APIs, and distributed systems. By addressing security risks like password exposure, enhancing scalability with stateless authentication, supporting Single Sign-On (SSO), and providing better session management, JWT has become the preferred method for handling authentication in many modern applications.

JWT Structure

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

```
Header.Payload.Signature
```

Each part is Base64Url encoded and serves a specific purpose.

1. Header

The header typically consists of two parts:

- **alg**: Specifies the signing algorithm (e.g., HS256, RS256, ES256).
- **typ**: Declares the token type as JWT.

Example:

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

After Base64Url encoding, it might look like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

2. Payload

The payload contains the claims, which are statements about an entity (typically, the user) and additional metadata.

Types of claims:

- **Registered Claims**: Predefined claims like `iss` (issuer), `exp` (expiration time), `sub` (subject), `aud` (audience), etc.
- **Public Claims**: Custom claims defined by users (e.g., user roles, email, etc.).

- **Private Claims**: Claims shared between parties that agree on their meaning.

Example:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1710336000
}
```

After Base64Url encoding:

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTcxMDMzNjAw
```

3. Signature

The signature ensures the integrity and authenticity of the token. It is generated by encoding the header and payload, then signing them with a secret key using the specified algorithm.

Example (using HMAC SHA-256):

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

If the secret key is `mysecretkey`, the resulting signature might look like:

```
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Final JWT

The final JWT is the concatenation of the three parts:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiw
```

JSON Web Token (JWT) Components

A **JSON Web Token (JWT)** is a compact, URL-safe token format used for authentication and secure data exchange between parties. It consists of three main components:

1. **Header**
2. **Payload**
3. **Signature**

Each of these components is Base64Url encoded and concatenated using a dot (`.`) separator.

---

1. Header

The **header** contains metadata about the token, including the algorithm used to sign it and the type of token.

Header Parameters

`alg` (Algorithm)

- Specifies the algorithm used for signing the token.
- Common values: `HS256`, `RS256`, `ES256`.

`typ` (Type)

- Identifies the token type, typically set to `JWT`.

Example Header JSON

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

Base64 Encoding

Once encoded using Base64Url encoding, the above JSON results in:

```
eyJhbGciOiAiSFMyNTYiLCAidHlwIjogIkpXVCJ9
```

---

2. Payload

The **payload** contains claims, which are statements about the entity (usually a user) and additional metadata.

Payload Claims

Reserved Claims

- Predefined claims with specific meanings.
- Examples:
    - `iss` (Issuer): Identifies the entity that issued the token.
    - `sub` (Subject): Identifies the subject of the token.
    - `exp` (Expiration Time): Specifies when the token expires.

Example Reserved Claims JSON

```
{
  "iss": "auth.example.com",
  "sub": "user123",
  "exp": 1711200000
}
```

Public Claims

- Custom claims that are publicly defined and commonly used.
- Example:

```
{
"role": "admin",
"permissions": ["read", "write", "delete"]
}
```

Private Claims

- Custom claims agreed upon between parties.
- Example:

```
{
"user_id": "abc-123",
"department": "engineering"
}
```

Base64 Encoding

Encoding the payload:

```
eyJpc3MiOiAiYXV0aC5leGFtcGxlLmNvbSIsICJzdWIiOiAidXNlcjEyMyIsICJleHAiOiAxNzExMjAwMDAwfQ
```

---

3. Signature

The **signature** ensures that the token has not been tampered with.

Signing Algorithms

HMAC (Hash-based Message Authentication Code)

- Uses a shared secret key.
- Example algorithm: `HS256`.

RSA (Rivest-Shamir-Adleman)

- Uses a public-private key pair.
- Example algorithm: `RS256`.

ECDSA (Elliptic Curve Digital Signature Algorithm)

- Uses elliptic curve cryptography for smaller key sizes.
- Example algorithm: `ES256`.

Verification Process

1. Decode the JWT.
2. Verify the signature using the appropriate algorithm and key.
3. Validate claims (e.g., expiration time, issuer).

JWT Usage

JSON Web Token (JWT) Overview

JWT (JSON Web Token) is a compact, URL-safe means of representing claims securely between two parties. JWTs are commonly used for authentication and information exchange in web applications and APIs. A JWT consists of three parts:

1. **Header**: Contains metadata about the token, such as the signing algorithm.
2. **Payload**: Holds the claims, which can include user details and additional metadata.
3. **Signature**: Used to verify that the token has not been tampered with.

JWTs are encoded using Base64URL and signed using either an HMAC secret or an asymmetric key pair.

Generating JWT

Generating a JWT involves the following steps:

1. **Construct the Header**: Defines the type of token (`JWT`) and the algorithm used for signing (e.g., `HS256`, `RS256`).
2. **Define the Payload**: Contains the claims, such as user identifiers, expiration, roles, and other metadata.
3. **Sign the Token**: Uses a secret key or private key to generate the signature.
4. **Encode the JWT**: The header, payload, and signature are concatenated using dots (`.`) to form the final token.

Example JWT Structure (JSON):

```
{
  "header": {
    "alg": "HS256",
    "typ": "JWT"
  },
  "payload": {
    "sub": "1234567890",
    "name": "John Doe",
    "iat": 1710000000,
    "exp": 1710600000
  },
  "signature": "HMACSHA256(Base64UrlEncode(header) + "." + Base64UrlEncode(payload), secret)"
}
```

Sending JWT in HTTP Requests

Once generated, a JWT must be included in HTTP requests to authenticate users. The most common method is via the `Authorization` header using the `Bearer` scheme.

Example HTTP Request with JWT:

```
{
  "method": "GET",
  "url": "https://api.example.com/protected-resource",
  "headers": {
    "Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiw

  }
}
```

Validating JWT

JWT validation ensures that the token is legitimate, has not been tampered with, and is still valid. Validation includes:

1. **Checking the Signature**: Verifying the token's signature using the shared secret (HMAC) or public key (RSA/ECDSA).
2. **Decoding the Token**: Extracting and reading the payload.
3. **Verifying Claims**: Ensuring mandatory claims like `iss` (issuer), `aud` (audience), and `sub` (subject) match expected values.
4. **Checking Expiration**: Ensuring the token has not expired (`exp` claim).

Example JWT Validation (JSON Response):

```
{
  "isValid": true,
  "claims": {
    "sub": "1234567890",
    "name": "John Doe",
    "iat": 1710000000,
    "exp": 1710600000
  },
  "signatureVerified": true,
  "issuerVerified": true,
  "expired": false
}
```

JWT Expiration

Expiration (`exp` claim) defines when a JWT becomes invalid. Tokens should have a reasonable lifespan to minimize security risks. If a token expires, clients need to obtain a new one through a refresh mechanism or re-authentication.

Example Expired JWT Response (JSON):

```
{
  "error": "TokenExpiredError",
  "message": "The provided JWT has expired",
  "expiredAt": 1710600000
}
```

Best Practices for Handling Expiration:

- Use short-lived tokens with refresh tokens for long-term access.
- Implement token rotation to mitigate replay attacks.
- Ensure secure storage and transmission of JWTs.
