What are JSON Web Tokens (JWTs)?

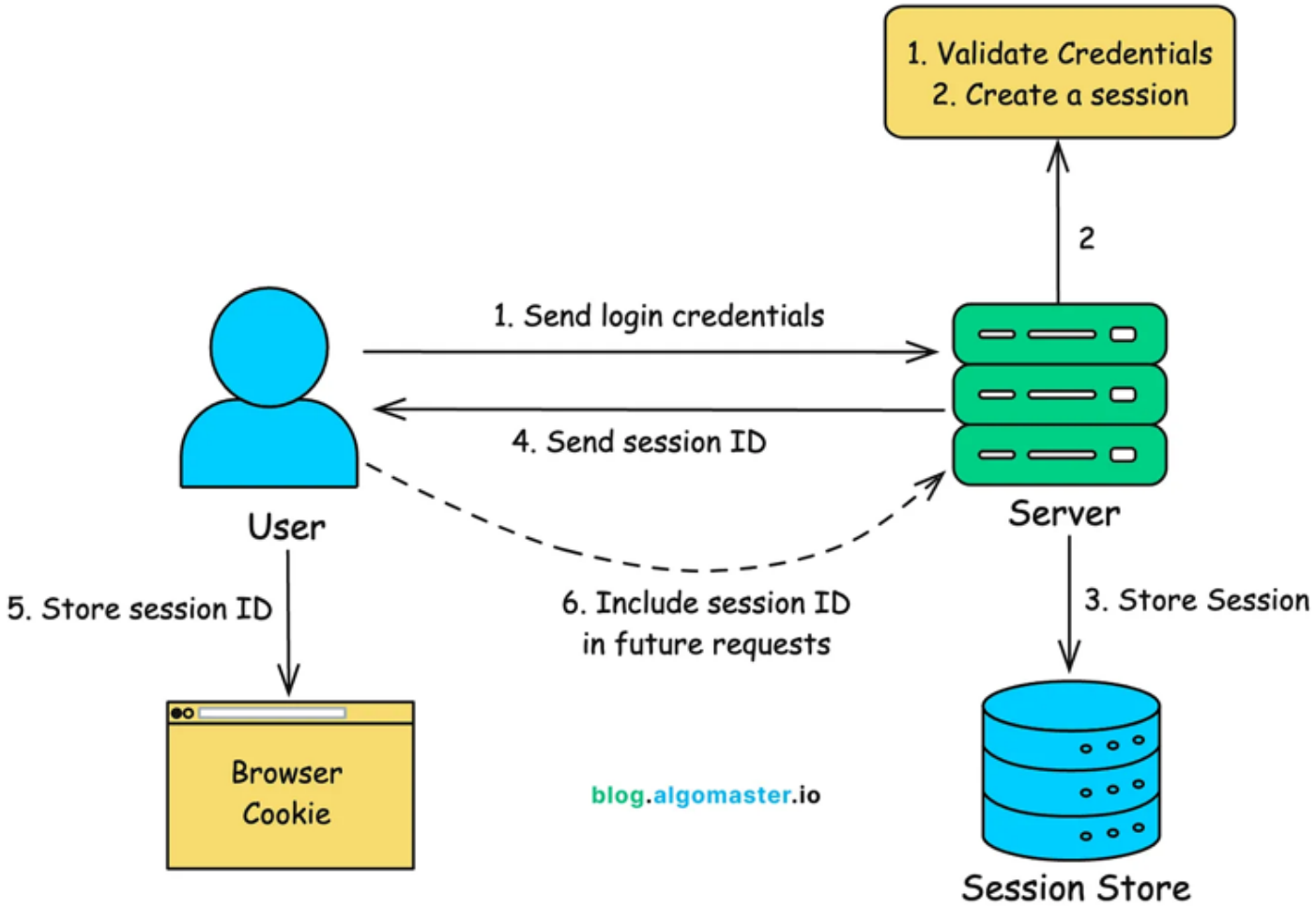
ASHISH PRATAP SINGH

JUN 10, 2025

A **JSON Web Token** (**JWT**) is an open standard for securely authenticating users and sending information between a client (like a web or mobile app) and a server.

In traditional web applications, user authentication relied on **server-side sessions**.

Here's how it typically worked:



1. The user logs in with their credentials.
2. The server validates those credentials and creates a session (with session ID).
3. Session data (like user ID and roles) is stored on the server, either in memory or a

database.

4. The server sends the session ID to the client .
5. The client stores the session ID locally (usually in a browser cookie).
6. For every subsequent request, the client sends this session ID.
7. The server looks up the session ID to authenticate and authorize the user.

While this approach is simple and effective for small applications, it doesn't scale well in **distributed systems or microservices architectures** since it requires maintaining shared session state across servers.

JWT-based authentication takes a different approach. It stores user claims directly inside the token, which remains on the client. The server only needs to verify the **token's signature** to authenticate the user, without storing any session data.

This makes them a great choice for building scalable APIs that do not depend on shared memory or server-side sessions.

In this article, we'll cover:

- What is JWT?
- Why do we need JWT?

- What a JWT token looks like?
- How does JWT work?
- Security considerations and best practices

If you're enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

---

1. What is a JWT?

> A **JWT** (**JSON Web Token**) is a compact, URL-safe token used to securely transmit information between two parties, typically a client and a server.

It is digitally signed, ensuring the data inside hasn't been tampered with and can be trusted. It's like a **digital ID card** that proves who you are.

JWTs are widely used in modern web and mobile applications, especially for **authentication** and **authorization**.

Here's why they're so popular:

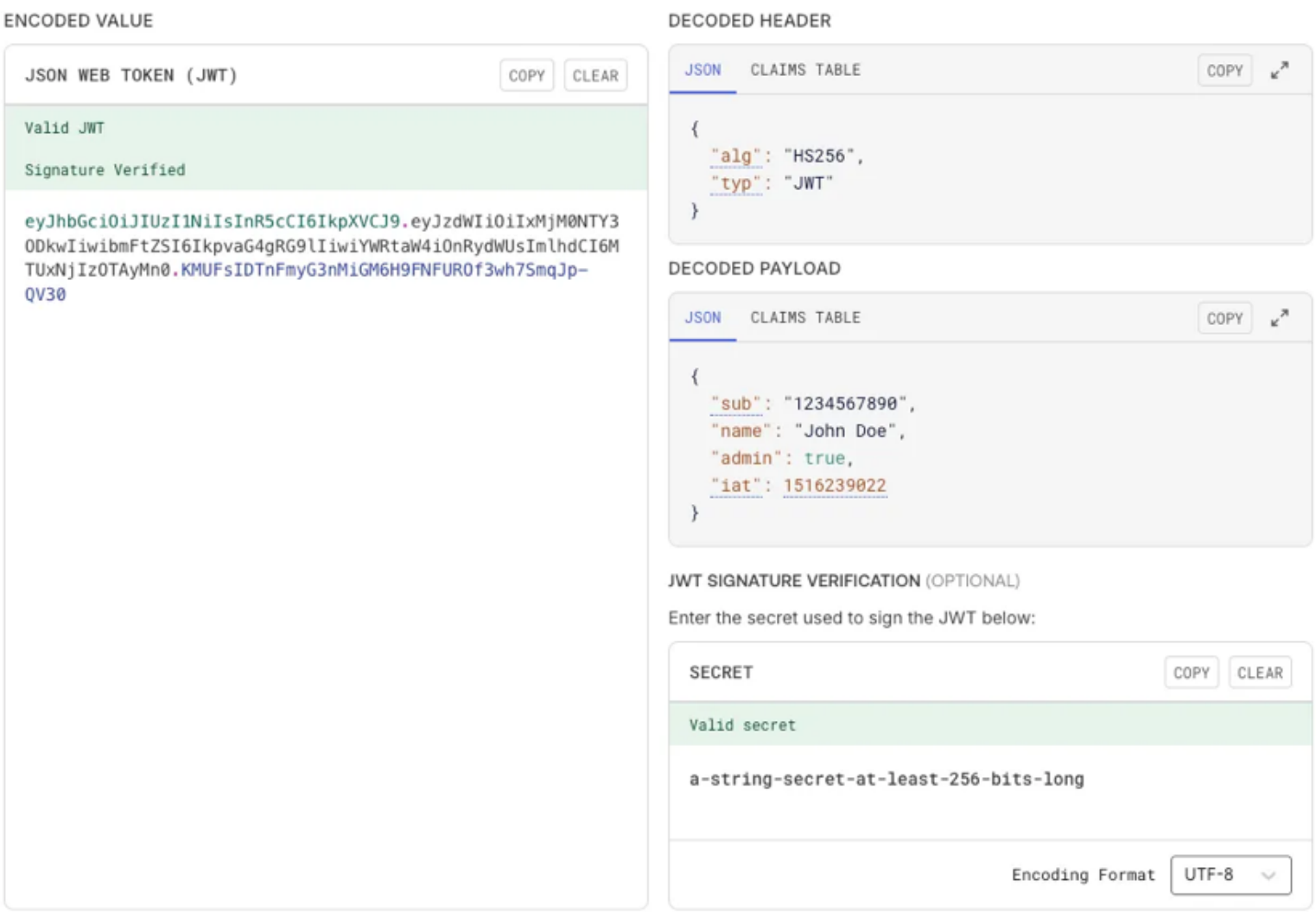
- **Stateless Authentication:** With JWTs, all the necessary user information (like user ID, role, and expiration time) is stored **within the token itself**.
- **Highly Scalable:** Because JWTs are **self-contained**, they can be easily used in distributed systems and microservices without the need to share session state across servers.
- **Secure by Design:** JWTs are signed (and optionally encrypted), ensuring that the data hasn't been tampered with and can be trusted.
- **Flexible and Extensible:** JWTs support **custom claims**, which means you can store any data your application needs such as roles, permissions, or tenant IDs.
- **Performance:** Verifying a signature (especially HMAC) can be faster than making a database call to look up session information.
- **Cross-Domain Communication (CORS):** Since JWTs are typically sent in an HTTP header, they work well across different domains, unlike cookies which have stricter domain policies.

---

2. The Anatomy of a JWT

A JSON Web Token (JWT) is made up of **three parts**, separated by dots:

```
header.payload.signature
```



Source: https://jwt.io

Each part serves a specific purpose. Let’s break it down.

1. Header

The header contains metadata about the token.

It usually looks like this:

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

- `alg`: The algorithm used to sign the token (e.g., HS256 for HMAC, RS256 for RSA).
- `typ`: The token type, always set to "JWT".

This JSON is then **Base64Url encoded** to form the **first part** of the token.

2. Payload

The payload contains the **claims**, information about the user or system. Claims are just pieces of data.

There are three types of claims:

a. Registered Claims

A set of predefined and commonly used claims. Not required, but recommended.

Examples:

- `iss`: Issuer (who issued the token)
- `sub`: Subject (who the token refers to, often the user ID)
- `aud`: Audience (who the token is intended for)
- `exp`: Expiration time (when the token expires)
- `nbf`: Not before (token becomes valid after this time)
- `iat`: Issued at (when the token was created)
- `jti`: Token ID (unique identifier to prevent reuse)

b. Public Claims

These are custom claims that are shared publicly. Should be unique to avoid conflicts, ideally defined in a public namespace.

c. Private Claims

These are custom claims created to share information between parties that agree on

using them. They are neither registered nor public claims. (e.g., userId, username, roles, permissions).

Example payload:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022,
  "exp": 1516242622 // Expires in 1 hour
}
```

This JSON is also **Base64Url encoded** to form the **second part** of the token.

> **Note:** The payload is encoded, not encrypted. Anyone can decode it, so **never put sensitive information like passwords inside**.

3. Signature

This is the security part. It's 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.

Here's how it's created:

1. Take the encoded header
2. Add a dot (`.`)
3. Add the encoded payload
4. Sign the resulting string using the algorithm mentioned in the header (`alg`) and a secret or private key

Example using HMAC SHA-256:

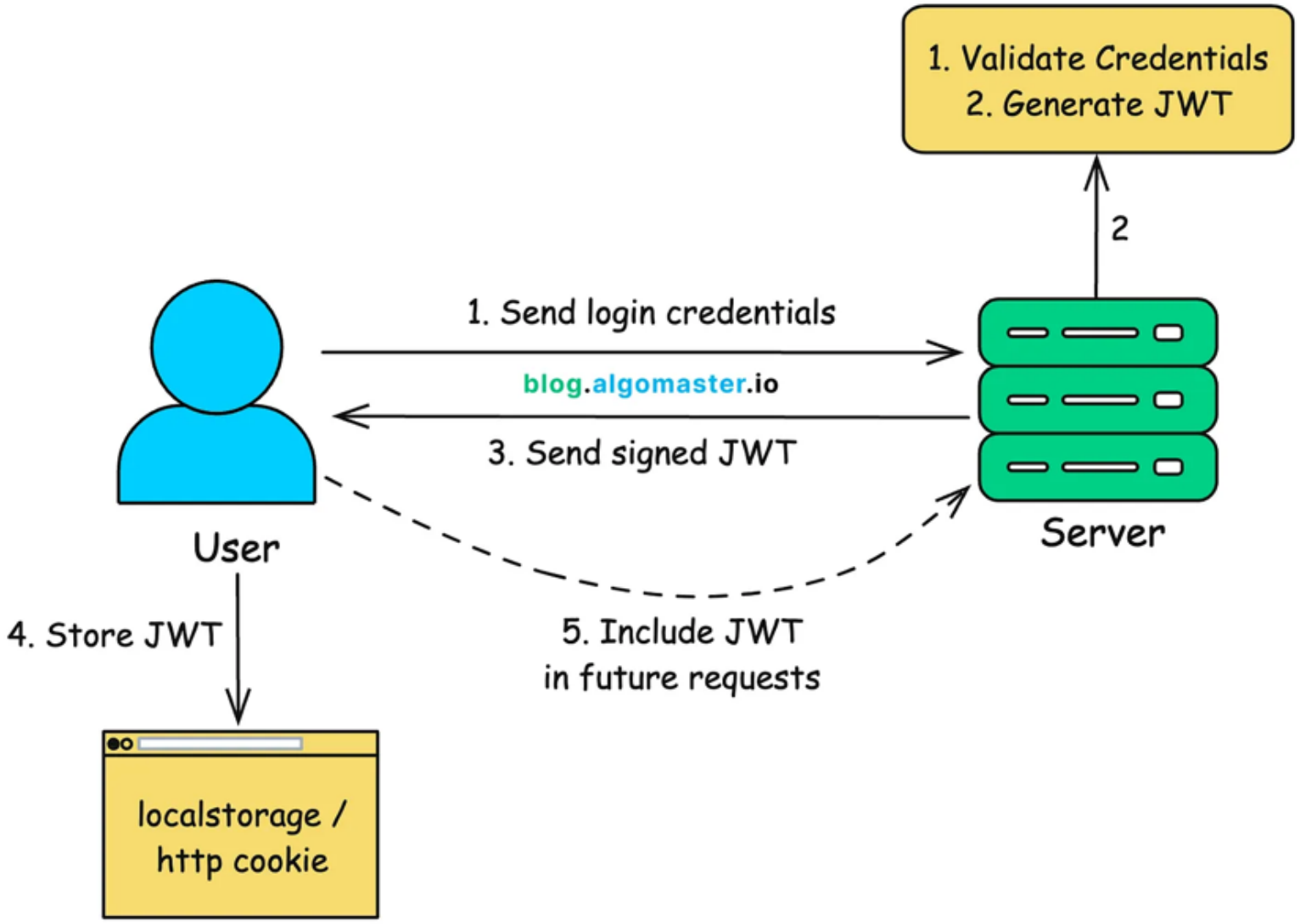
```
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  secret
)
```

The result is **Base64Url encoded** and forms the **third part** of the token.

---

3. How JWT Works

Here's a step-by-step look at how authentication typically works using JWTs:



Step 1: User Logs In

The user initiates a login by sending a POST request with their email and password or social login token:

```
POST /login
{
  "email": "test@example.com",
  "password": "mypassword"
}
```

Step 2: Server Verifies Credentials & Generates JWT

If the credentials are valid, the authentication server generates a JWT:

```
const token = jwt.sign(
  { userId: 123, role: 'admin' },
  SECRET_KEY,
  { expiresIn: '1h' }
);
```

- The token payload includes user-specific claims, such as user ID and role.
- The token also contains an expiration time (`exp`).
- The header and payload are signed using a secret key (for HS256) or a private key (for RS256).

The server sends the generated JWT back to the client, typically in the response body.

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR..."
}
```

Step 3: Client Stores JWT

The client securely stores the token for use in future requests. Common storage options include:

- **Web applications**: `localStorage` or `sessionStorage` (though these are vulnerable to XSS attacks), or preferably an **HttpOnly cookie** (which cannot be accessed via JavaScript).
- **Mobile apps**: Platform-specific secure storage solutions, such as Keychain on iOS

or Keystore on Android.

Step 4: Client Sends JWT with Each Request

For every subsequent API call, the client includes the JWT in the `Authorization` header using the Bearer schema:

```
GET /dashboard
Authorization: Bearer <JWT_TOKEN>
```

This allows the server to identify and authenticate the user without storing session state.

Step 5: Server Verifies Signature and Grants Access

The server:

- Extracts the token from the `Authorization` header
- Verifies the token's signature and checks for expiration

If the token is valid, the user is authenticated and the server processes the request.

If the token is invalid or expired, the server responds with an appropriate error, such as:

```
401 Unauthorized
403 Forbidden
```

---

5. Security Considerations and Best Practices

While JWTs are powerful, incorrect implementation can lead to serious vulnerabilities.

Here's how to use them securely:

Always Use HTTPS

Transmit JWTs only over HTTPS to prevent them from being intercepted in transit.

Keep Secrets/Private Keys Secure

If the secret is leaked, tokens can be forged.

- For **HS256**, use strong, randomly generated secret keys.
- For **RS256**/**ES256**, protect the **private key** carefully. Never expose it on the client or in version control.

Choose Strong Signing Algorithms

- Avoid `alg: "none"` at all costs. Some older libraries allowed this, disabling signature checks. Make sure your library **rejects** tokens with `alg: none`.
- Choose strong algorithms like **HS256**, **RS256**, or **ES256.** Avoid older/weaker ones.

Always Verify the Signature

This is the core of JWT security. Never trust a token without validating its signature.

Validate Standard Claims

These claims are essential for ensuring a token is legitimate and appropriate for your service:

- `exp` (**Expiration Time**) – Always set and validate it. Short-lived tokens are generally more secure.
- `nbf` (**Not Before**) – Optional, but if present, should be checked.

- `aud` (**Audience**) – Ensure the token was meant for your application.
- `iss` (**Issuer**) – Confirm the token was issued by a trusted source.

Don’t Include Sensitive Data in the Payload

JWT payloads are **Base64Url-encoded, not encrypted**. Anyone can decode and read them. Never include secrets, passwords, or sensitive personal data.

Handle Token Expiration and Revocation

JWTs are stateless, which means once issued, they can’t be revoked easily. You must design with that in mind.

Recommended strategies:

- **Short Expiration Times** – Issue access tokens that expire quickly (e.g., 15 minutes).
- **Refresh Tokens** – Use longer-lived refresh tokens to issue new access tokens. Refresh tokens can be revoked if needed.
- **Token Blacklists** – Keep a list of revoked tokens or JTIs. This reintroduces server-side state but allows targeted revocation.
- **Replay Protection** – Use the `jti` claim to uniquely identify tokens and track their

usage (optional and stateful).

Secure Storage on the Client

- **localStorage/sessionStorage** – Easy but vulnerable to **XSS attacks**.
- **HttpOnly, Secure Cookies** – Safer from XSS, but requires additional care for **CSRF** protection.

Best practice for web apps: Use **access tokens in memory** and **refresh tokens in HttpOnly cookies** to balance security and usability.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

This post is public so feel free to share it.

P.S. If you're enjoying this newsletter and want to get even more value, consider

becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

There are group discounts, gift options, and referral bonuses available.

Checkout my Youtube channel for more in-depth content.

Follow me on LinkedIn and X to stay updated.

Checkout my GitHub repositories for free interview preparation resources.

I hope you have a lovely day!

See you soon,

Ashish

178 Likes • 10 Restacks

← Previous Next →

Discussion about this post

Comments Restacks

Write a comment...

Punitha Veeramani 10 Jun

Liked by Ashish Pratap Singh

Its useful. Timely helped me to clarify few things. Thank you

LIKE (2) REPLY SHARE

1 reply

Krishna 22 Jun

Liked by Ashish Pratap Singh

Excellent article with good visual representation !!

LIKE (1) REPLY SHARE

7 more comments...