JWT Authentication & Authorization

A Complete Guide for Developers

1. JWT Basics: Self-Contained & Stateless

JWT (JSON Web Token) is a compact, URL-safe method for securely transmitting information between parties. It's self-contained, meaning all necessary user information is embedded within the token itself, eliminating the need for server-side session storage. This makes JWTs inherently stateless.

Its structure consists of three Base64URL-encoded parts, separated by dots:

- **Header**: Describes the token type and algorithm.
- **Payload**: Contains claims (statements about an entity, e.g., user data).
- **Signature**: Verifies the token's integrity and authenticity.

Example JWT Token

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

2. JWT Structure Deep Dive

Header (Base64URL Encoded)

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

Specifies the signing algorithm (e.g., HMAC SHA256) and the token type, which is always "JWT". This part is crucial for the receiving party to understand how to verify the token.

Payload (Base64URL Encoded)

`{ "sub": "1234567890", "name": "John Doe", "role": "admin", "exp": 1735689600 }`

Contains claims, which are statements about the user and additional data. These can be registered claims (standardized, like `sub`, `name`, `exp`), public claims (custom claims registered in IANA JWT Registry), or private claims (custom claims agreed upon by parties).

Signature

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

Created by taking the encoded header, the encoded payload, and a secret key. This signature ensures that the token hasn't been tampered with. If the header or payload is modified, the signature will no longer be valid, and the token will be rejected.

3. Creating a JWT

Tokens are generated by the server upon successful user authentication. The `jsonwebtoken` library in Node.js is commonly used for this purpose. The payload typically includes unique user identifiers and any relevant roles or permissions, while the secret key ensures the token's integrity. It's crucial to set an expiration time to enhance security.

Best Practice: Store your `JWT_SECRET` in environment variables (`process.env.JWT_SECRET`) to prevent hardcoding and ensure it remains confidential, especially in production environments.

Example: Server-Side Token Generation

```
const jwt = require('jsonwebtoken');

function createToken(user) {
 const payload = {
 userId: user.id,
 email: user.email,
 role: user.role
 };
 return jwt.sign(payload, process.env.JWT_SECRET, {
expiresIn: '1h' });
}

// Usage:
// const token = createToken(user);
// res.json({ token });
```

4. Verifying JWT

On protected routes, the server must verify the authenticity and validity of the JWT presented by the client. This typically happens via a middleware function that intercepts requests.

The verification process involves checking the token's signature against the server's secret key. If verification fails (e.g., token is invalid, expired, or tampered with), access is denied. If successful, the decoded user information is attached to the request object for subsequent use in the route handler.

Note: Tokens are commonly sent in the `Authorization` header with the "Bearer" scheme (e.g., "Bearer YOUR_TOKEN_HERE").

Example: Server-Side JWT Verification Middleware

```
function verifyToken(req, res, next) {
 const token = req.headers['authorization']?.split(' ')[1];

 if (!token) {
 return res.status(401).json({ error: 'Access denied' });
 }

 try {
 const decoded = jwt.verify(token, process.env.JWT_SECRET);
 req.user = decoded; // Attach user info to request
 next();
 } catch (error) {
 res.status(403).json({ error: 'Invalid token' });
 }
}
```

5. JWT vs. Sessions: Stateless vs. Stateful

Traditional Sessions (Stateful)

```
app.post('/login', (req, res) => {
  // Validate user, create session
  req.session.userId = user.id; // Stored on server
  res.json({ message: 'Logged in' });
});
```

- **Server-side Storage:** Session data stored in server memory or database.
- **Scalability:** Can be challenging to scale horizontally (sticky sessions or shared storage).
- **Invalidation:** Easy to invalidate a specific session by deleting it on the server.

JWT (Stateless)

```
app.post('/login', (req, res) => {
  // Validate user, create token
  const token = jwt.sign({ userId: user.id }, secret);
  res.json({ token }); // Client stores token
});
```

- **Client-side Storage:** Token stored on the client (e.g., localStorage, cookies).
- **Scalability:** Highly scalable; no server state to manage for authentication.
- **Invalidation:** Tokens expire, but immediate invalidation (e.g., for logout) is harder (requires blacklisting).

6. Common JWT Claims

1

Registered Claims

Standardized, interoperable claims defined in the JWT specification. These are optional but recommended for general use.

- **iss (Issuer):** Identifies the principal that issued the JWT (e.g., `myapp.com`).
- **sub (Subject):** Identifies the principal that is the subject of the JWT (e.g., `user123`).
- **aud (Audience):** Identifies the recipients that the JWT is intended for (e.g., `myapp-users`).
- **exp (Expiration Time):** NumericDate value indicating when the JWT expires (`Math.floor(Date.now() / 1000) + 3600`).
- **iat (Issued At):** NumericDate value indicating when the JWT was issued (`Math.floor(Date.now() / 1000)`).

2

Public Claims

Custom claims meant to be publicly registered in the **IANA JSON Web Token Claims Registry** to avoid collisions. They are less common in typical applications but can be useful for specific interoperability needs.

3

Private Claims

Custom claims created for specific applications or agreements between parties. These are not registered and should be used with caution to avoid name collisions, though in practice, they are widely used.

- **email:** `user@example.com`
- **role:** `admin`
- **permissions:** ['read', 'write']

7. JWT in Authentication Flow

Backend: Login Endpoint

```
app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  // 1. Validate credentials (e.g., check against DB)
  const user = await User.findOne({ email });
  const isValid = await bcrypt.compare(password,
user.password);

  if (isValid) {
    // 2. Generate JWT upon successful validation
    const token = jwt.sign(
      { userId: user.id, role: user.role },
      process.env.JWT_SECRET,
      { expiresIn: '24h' }
    );
    // 3. Send token to client
    res.json({ token, user: { id: user.id, email } });
  } else {
    res.status(401).json({ error: 'Invalid credentials' });
  }
});
```

Frontend: Client-Side Token Storage

```
const login = async (email, password) => {
  const response = await fetch('/login', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ email, password })
  });
  const data = await response.json();
  // Store token (e.g., in localStorage)
  localStorage.setItem('token', data.token);
  // Redirect or update UI
};
```

After receiving the token, the client (e.g., a web browser or mobile app) is responsible for securely storing it. `localStorage` is a common choice, but be mindful of XSS attacks. For enhanced security, consider HTTP-only cookies, especially for web applications.

8. Authorization with JWT

JWTs facilitate authorization by carrying user roles and permissions in their payload. After token verification, this information can be used to control access to specific API routes or resources.

A common pattern is to implement an authorization middleware that checks the `req.user.role` (decoded from the JWT) against a list of allowed roles for a given route. If the user's role does not match, a `403 Forbidden` status is returned.

Granular Permissions: Instead of just roles, you can include specific permissions (e.g., `"can_delete_users"`) in the JWT payload for finer-grained access control.

Example: Role-Based Authorization Middleware

```
function authorize(roles = []) {
 return (req, res, next) => {
 const userRole = req.user.role; // Role from decoded JWT

 if (roles.length && !roles.includes(userRole)) {
 return res.status(403).json({ error: 'Insufficient permissions'
});
 }
 next();
 };
}

// Usage:
app.get('/admin', verifyToken, authorize(['admin']), (req, res)
=> {
 res.json({ message: 'Admin only content' });
});

app.get('/user', verifyToken, authorize(['user', 'admin']), (req,
res) => {
 res.json({ message: 'User content' });
});
```

9. JWT Security Best Practices

Strong Secret Key

Always use a long, complex, and cryptographically strong secret key (at least 32 bytes/256 bits). Generate it securely (e.g., `crypto.randomBytes(64).toString('hex')`) and store it in environment variables.

Appropriate Expiration Times

Set short expiration times (e.g., 15 minutes to 1 hour) for access tokens to minimize exposure if a token is compromised. Use refresh tokens for longer validity periods, as discussed next.

Always Use HTTPS

Ensure all communication, especially token transmission, occurs over HTTPS (SSL/TLS) to prevent man-in-the-middle attacks where tokens could be intercepted.

Secure Storage on Client-Side

For web applications, storing JWTs in HTTP-only cookies can mitigate XSS attacks. If using `localStorage`, implement robust XSS protection on your frontend.

Validate Token Structure & Claims

Beyond signature verification, always validate standard claims like `iss`, `aud`, and `exp` to ensure the token is for your application and hasn't expired.