Engineer

JWT

a compact, URL-safe token that carries claims as JSON

What it is

A JWT is a short string that carries a set of statements from one party to another in a form the receiver can check. The letters stand for JSON Web Token. RFC 7519 defined it in May 2015 as a compact way of representing claims that is safe to put in a URL, and notes that the name is pronounced “jot”. A claim is a name and a value asserted about a subject: `"sub": "24400320"` says whom the token is about, `"exp": 1300819380` says when it stops being valid.

On the wire a JWT is a series of base64url strings joined by dots. A signed one has three: a header that names the algorithm, a payload that holds the claims, and a signature computed over the first two. An encrypted one has five, and its claims cannot be read without the key. RFC 7519 does none of the cryptography itself. It relies on JSON Web Signature (RFC 7515) for tokens that are signed or protected with a message authentication code, and on JSON Web Encryption (RFC 7516) for tokens that are encrypted; the algorithms are listed in RFC 7518 and the keys can be published in the JSON Web Key format of RFC 7517.

`HS256` is an HMAC with SHA-256: one shared secret both creates and checks the token, so every service that can verify it can also mint it. `RS256` and `ES256` use a key pair: the issuer signs with a private key and everyone else verifies with the public one, which can be handed out freely. RFC 7519 makes only `HS256` and the unsigned `none` mandatory to implement, and recommends `RS256` and `ES256`.

Section 4.1 registers seven claim names: `iss` (issuer), `sub` (subject), `aud` (audience), `exp` (expiration time), `nbf` (not before), `iat` (issued at) and `jti` (JWT ID, a unique identifier that can be used to stop a token being replayed). They are three letters long because the format is meant to be compact, and every one of them is optional. Which claims a token must carry is left to the

application that uses it. Beyond the registered set, anyone can define public claims, either registered with IANA or given a name that cannot collide with anyone else's, such as a URL, or private claims agreed between two parties.

Use the concept in a decision

Put "What is JWT?" to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

The point of a JWT is that the receiver can check it on its own. With the issuer's key and a clock, a service can tell whether the token is genuine, whom it is for and whether it is still valid, without a database lookup and without calling the issuer. RFC 9068 (October 2021) records that many commercial OAuth implementations issue access tokens that resource servers parse and validate directly, a pattern common where the server that issues tokens and the one that accepts them are separated by some boundary, and that many of them use JWT. It then standardised the layout, so that such tokens work across vendors.

It is also the format behind OpenID Connect: every ID token is a JWT addressed to the application that asked for it. The provider must sign it unless both conditions for using `alg` set to `none` hold: the response type returns no ID token from the Authorization Endpoint, as in the Authorization Code Flow, and the client explicitly requested `none` at registration. When the media type `application/jwt` was registered in 2015, the registration already named OpenID Connect, Salesforce, Google, Android, Windows Azure and Amazon Web Services among its users.

RFC 7519 aims the format at space-constrained places including the Authorization header and parameters in a URI's query, which is why the claim names are short and the encoding passes through a URL unescaped.

The same property that makes a JWT cheap to check makes it hard to take back. A receiver that verifies the signature and the clock will accept the token until `exp`, whatever has happened to the account in the meantime.

What it looks like in practice

RFC 7519 works through a complete example. The header declares that the object is a JWT and that it is protected with HMAC SHA-256:

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

The claims name an issuer, an expiry time and one public claim whose name is a URL, so that it cannot clash with a claim anyone else defines:

```
{"iss":"joe",
 "exp":1300819380,
 "http://example.com/is_root":true}
```

Each part is encoded with base64url, the HMAC is computed over the header and payload joined by a dot, and the three results are joined into one string. The RFC prints it with line breaks for display only:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Three things follow from this example. First, the claims are readable by anyone who holds the token: base64url is an encoding, so decoding the middle part gives the JSON back without any key. Second, the signature covers the encoded strings exactly as they were sent. The RFC's header contains a CRLF line break and a leading space, and it lists the exact bytes for that reason; formatting the JSON more neatly and encoding it again would produce a different string, and the signature would no longer match. The key for this example is printed in Appendix A.1 of RFC 7515, and with it the signature verifies. Third, the token has long expired: `1300819380` seconds after 1 January 1970 is 22 March 2011 at 18:43 UTC, so a correct validator rejects it today.

A receiver that gets a token like this runs a fixed set of checks before it reads a single claim. It splits the string at the dots, decodes the header and confirms that it is valid JSON containing only parameters and values whose syntax and meaning it understands and supports, or that are explicitly specified to be ignored when unknown. It compares `alg` with the algorithms it has decided to accept for this issuer and key: RFC 7519 is explicit that this is an application decision, and that a token should be rejected if its algorithm is unacceptable even when the signature is valid. It verifies the signature with a key that belongs to the issuer. Then it checks the claims: `iss` must be an issuer it trusts, `aud` must name the receiver itself, the current time must be before `exp`, allowing a few minutes at most for clock skew, and after `nbf` if that claim is present. Only then does it act on the claims.

A JWT access token under RFC 9068 shows what a real profile adds. Its header in that RFC reads `{"typ":"at+JWT","alg":"RS256","kid":"RjEwOwOA"}`: an explicit type, an asymmetric algorithm, and the ID of the key in the issuer's published set. The profile makes `iss`, `exp`, `aud`, `sub`, `client_id`, `iat` and `jti` mandatory, and a resource server must check that `typ` is `at+jwt` or `application/at+jwt` and reject other types. These media type and subtype names are case-insensitive; `at+JWT` in the example is an equivalent spelling.

The claims inside a token are a snapshot taken when it was signed, so anything that can change before `exp` is better decided on the server. One of this site's projects, hardening an application at every layer, re-checks the entitlements (tier, role, organisation and ship scoping) on the server on every request and treats the gates in the browser as UX only.

Where it goes wrong

The token picks its own algorithm. The `alg` header is there for flexibility, and RFC 8725 describes how some libraries trusted it too far. An attacker could change it to `none` and the library would accept the token with no signature at all; or change `RS256` to `HS256`, and the library would check an HMAC using the issuer's public RSA key as the shared secret, a key the attacker can download. Configure the library with the algorithms you accept, use each key with exactly one algorithm, and reject `none` outright.

The HMAC secret is a password. A token signed with `HS256` under a short, memorable secret can be attacked offline: anyone holding a single token can try guesses against it at their own pace, without touching your server. RFC 8725

forbids using a human-memorable password directly as an HMAC key, and RFC 7518 requires an `HS256` key of at least 256 bits. Generate the key from a cryptographic random source, keep it in a secret store, and prefer a key pair when many services only need to verify.

The signature checks out, so everything is believed. A valid signature proves who issued the token, not whom it was for. If one issuer serves several services and the receiver skips `aud`, a token legitimately sent to one service can be replayed against another. RFC 8725 calls this substitution and requires any receiver of such tokens to validate the audience and reject a token that lacks one. Check `aud` against your own identifier, and check that the signing key belongs to the issuer named in `iss`.

One kind of token passes for another. An ID token, an access token and a security event token from the same issuer can look much alike, and a receiver that only checks signature and issuer may accept the wrong kind. RFC 8725 calls this cross-JWT confusion and recommends explicit typing through the `typ` header; RFC 9068 applies it with `at+jwt`. Give each kind of token its own type or audience, and write the validation rules so that each kind fails the others' checks.

The header chooses the key. The `kid` header says which key to use, and `jku` and `x5u` can point to a URL from which to fetch it. All of them arrive from the sender. A `kid` passed unchecked into a database query or a directory lookup becomes a way into that system, and a server that follows any URL it is given in `jku` or `x5u` can be made to send requests wherever the sender chooses. Look `kid` up only among keys you already hold, and fetch keys only from an allow-list of locations, such as the issuer's published `jwks_uri`.

Anyone can read the payload. A signed JWT protects its claims against tampering, not against reading, and it often travels through browsers, logs and proxies. RFC 7519 notes that a JWT may contain privacy-sensitive information and that the simplest protection is to leave it out; otherwise the token should be encrypted, or at least sent only over TLS. RFC 9068 adds that a client must not inspect an access token at all, since the issuer may change its format. Put identifiers in the token, not personal data, and encrypt when the content itself must stay private.

Further reading

- RFC 7519: JSON Web Token (JWT) — the format itself, with the registered claims, the worked example and the validation steps.
- RFC 7515: JSON Web Signature (JWS) — how the header, payload and signature are encoded and joined, and the key used in the example above.
- RFC 8725: JSON Web Token Best Current Practices — the known attacks on JWT and the rules for libraries and applications that prevent them.
- RFC 7518: JSON Web Algorithms (JWA) — the algorithm names used in `alg`, with the minimum key size for each HMAC.
- RFC 9068: JWT Profile for OAuth 2.0 Access Tokens — the standard layout for JWT access tokens and the checks a resource server must run.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is JWT?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/jwt/