[jsonwebtoken](https://www.npmjs.com/package/jsonwebtoken)

Top of Form

Bottom of Form

An implementation of [**JSON Web Tokens**](https://tools.ietf.org/html/rfc7519).

This was developed against draft-ietf-oauth-json-web-token-08. It makes use of [**node-jws**](https://github.com/brianloveswords/node-jws)

Install

$ npm install jsonwebtoken

Usage

**jwt.sign(payload, secretOrPrivateKey, [options, callback])**

(Asynchronous) If a callback is supplied, callback is called with the err or the JWT.

(Synchronous) Returns the JsonWebToken as string

payload could be an object literal, buffer or string. *Please note that* exp is only set if the payload is an object literal.

secretOrPrivateKey is a string or buffer containing either the secret for HMAC algorithms, or the PEM encoded private key for RSA and ECDSA.

options:

* algorithm (default: HS256)
* expiresIn: expressed in seconds or a string describing a time span **[zeit/ms](https://github.com/zeit/ms)**. Eg: 60, "2 days", "10h", "7d"
* notBefore: expressed in seconds or a string describing a time span **[zeit/ms](https://github.com/zeit/ms)**. Eg: 60, "2 days", "10h", "7d"
* audience
* issuer
* jwtid
* subject
* noTimestamp
* header

If payload is not a buffer or a string, it will be coerced into a string using JSON.stringify.

There are no default values for expiresIn, notBefore, audience, subject, issuer. These claims can also be provided in the payload directly with exp, nbf, aud, sub and iss respectively, but you can't include in both places.

Remember that exp, nbf and iat are **NumericDate**, see related [**Token Expiration (exp claim)**](https://www.npmjs.com/package/jsonwebtoken#token-expiration-exp-claim)

The header can be customized via the option.header object.

Generated jwts will include an iat (issued at) claim by default unless noTimestamp is specified. If iatis inserted in the payload, it will be used instead of the real timestamp for calculating other things like exp given a timespan in options.expiresIn.

Example

*// sign with default (HMAC SHA256)*

var jwt **=** require('jsonwebtoken');

var token **=** jwt.sign({ foo**:** 'bar' }, 'shhhhh');

*//backdate a jwt 30 seconds*

var older\_token **=** jwt.sign({ foo**:** 'bar', iat**:** Math.floor(Date.now() **/** 1000) **-** 30 }, 'shhhhh');

*// sign with RSA SHA256*

var cert **=** fs.readFileSync('private.key');  *// get private key*

var token **=** jwt.sign({ foo**:** 'bar' }, cert, { algorithm**:** 'RS256'});

*// sign asynchronously*

jwt.sign({ foo**:** 'bar' }, cert, { algorithm**:** 'RS256' }, function(err, token) {

  console.log(token);

});

**Token Expiration (exp claim)**

The standard for JWT defines an exp claim for expiration. The expiration is represented as a **NumericDate**:

A JSON numeric value representing the number of seconds from 1970-01-01T00:00:00Z UTC until the specified UTC date/time, ignoring leap seconds. This is equivalent to the IEEE Std 1003.1, 2013 Edition [POSIX.1] definition "Seconds Since the Epoch", in which each day is accounted for by exactly 86400 seconds, other than that non-integer values can be represented. See RFC 3339 [RFC3339] for details regarding date/times in general and UTC in particular.

This means that the exp field should contain the number of seconds since the epoch.

Signing a token with 1 hour of expiration:

jwt.sign({

  exp**:** Math.floor(Date.now() **/** 1000) **+** (60 **\*** 60),

  data**:** 'foobar'

}, 'secret');

Another way to generate a token like this with this library is:

jwt.sign({

  data**:** 'foobar'

}, 'secret', { expiresIn**:** 60 **\*** 60 });

*//or even better:*

jwt.sign({

  data**:** 'foobar'

}, 'secret', { expiresIn**:** '1h' });

**jwt.verify(token, secretOrPublicKey, [options, callback])**

(Asynchronous) If a callback is supplied, function acts asynchronously. Callback is passed the decoded payload if the signature and optional expiration, audience, or issuer are valid. If not, it will be passed the error.

(Synchronous) If a callback is not supplied, function acts synchronously. Returns the payload decoded if the signature (and, optionally, expiration, audience, issuer) are valid. If not, it will throw the error.

token is the JsonWebToken string

secretOrPublicKey is a string or buffer containing either the secret for HMAC algorithms, or the PEM encoded public key for RSA and ECDSA.

options

* algorithms: List of strings with the names of the allowed algorithms. For instance, ["HS256", "HS384"].
* audience: if you want to check audience (aud), provide a value here
* issuer (optional): string or array of strings of valid values for the iss field.
* ignoreExpiration: if true do not validate the expiration of the token.
* ignoreNotBefore...
* subject: if you want to check subject (sub), provide a value here
* clockTolerance: number of seconds to tolerate when checking the nbf and exp claims, to deal with small clock differences among different servers
* maxAge: the maximum allowed age for tokens to still be valid. Currently it is expressed in milliseconds or a string describing a time span **[zeit/ms](https://github.com/zeit/ms)**. Eg: 1000, "2 days", "10h", "7d". **We advise against using milliseconds precision, though, since JWTs can only contain seconds. The maximum precision might be reduced to seconds in the future.**
* clockTimestamp: the time in seconds that should be used as the current time for all necessary comparisons (also against maxAge, so our advise is to avoid using clockTimestamp and a maxAgein milliseconds together)

*// verify a token symmetric - synchronous*

var decoded **=** jwt.verify(token, 'shhhhh');

console.log(decoded.foo) *// bar*

*// verify a token symmetric*

jwt.verify(token, 'shhhhh', function(err, decoded) {

  console.log(decoded.foo) *// bar*

});

*// invalid token - synchronous*

**try** {

  var decoded **=** jwt.verify(token, 'wrong-secret');

} **catch**(err) {

*// err*

}

*// invalid token*

jwt.verify(token, 'wrong-secret', function(err, decoded) {

*// err*

*// decoded undefined*

});

*// verify a token asymmetric*

var cert **=** fs.readFileSync('public.pem');  *// get public key*

jwt.verify(token, cert, function(err, decoded) {

  console.log(decoded.foo) *// bar*

});

*// verify audience*

var cert **=** fs.readFileSync('public.pem');  *// get public key*

jwt.verify(token, cert, { audience**:** 'urn:foo' }, function(err, decoded) {

*// if audience mismatch, err == invalid audience*

});

*// verify issuer*

var cert **=** fs.readFileSync('public.pem');  *// get public key*

jwt.verify(token, cert, { audience**:** 'urn:foo', issuer**:** 'urn:issuer' }, function(err, decoded) {

*// if issuer mismatch, err == invalid issuer*

});

*// verify jwt id*

var cert **=** fs.readFileSync('public.pem');  *// get public key*

jwt.verify(token, cert, { audience**:** 'urn:foo', issuer**:** 'urn:issuer', jwtid**:** 'jwtid' }, function(err, decoded) {

*// if jwt id mismatch, err == invalid jwt id*

});

*// verify subject*

var cert **=** fs.readFileSync('public.pem');  *// get public key*

jwt.verify(token, cert, { audience**:** 'urn:foo', issuer**:** 'urn:issuer', jwtid**:** 'jwtid', subject**:** 'subject' }, function(err, decoded) {

*// if subject mismatch, err == invalid subject*

});

*// alg mismatch*

var cert **=** fs.readFileSync('public.pem'); *// get public key*

jwt.verify(token, cert, { algorithms**:** ['RS256'] }, function (err, payload) {

*// if token alg != RS256,  err == invalid signature*

});

**jwt.decode(token [, options])**

(Synchronous) Returns the decoded payload without verifying if the signature is valid.

**Warning:** This will **not** verify whether the signature is valid. You should **not** use this for untrusted messages. You most likely want to use jwt.verify instead.

token is the JsonWebToken string

options:

* json: force JSON.parse on the payload even if the header doesn't contain "typ":"JWT".
* complete: return an object with the decoded payload and header.

Example

*// get the decoded payload ignoring signature, no secretOrPrivateKey needed*

var decoded **=** jwt.decode(token);

*// get the decoded payload and header*

var decoded **=** jwt.decode(token, {complete**:** true});

console.log(decoded.header);

console.log(decoded.payload)

**Errors & Codes**

Possible thrown errors during verification. Error is the first argument of the verification callback.

**TokenExpiredError**

Thrown error if the token is expired.

Error object:

* name: 'TokenExpiredError'
* message: 'jwt expired'
* expiredAt: [ExpDate]

jwt.verify(token, 'shhhhh', function(err, decoded) {

**if** (err) {

*/\**

*err = {*

*name: 'TokenExpiredError',*

*message: 'jwt expired',*

*expiredAt: 1408621000*

*}*

*\*/*

  }

});

**JsonWebTokenError**

Error object:

* name: 'JsonWebTokenError'
* message:
  + 'jwt malformed'
  + 'jwt signature is required'
  + 'invalid signature'
  + 'jwt audience invalid. expected: [OPTIONS AUDIENCE]'
  + 'jwt issuer invalid. expected: [OPTIONS ISSUER]'
  + 'jwt id invalid. expected: [OPTIONS JWT ID]'
  + 'jwt subject invalid. expected: [OPTIONS SUBJECT]'

jwt.verify(token, 'shhhhh', function(err, decoded) {

**if** (err) {

*/\**

*err = {*

*name: 'JsonWebTokenError',*

*message: 'jwt malformed'*

*}*

*\*/*

  }

});

**Algorithms supported**

Array of supported algorithms. The following algorithms are currently supported.

| **alg parameter value** | **digital signature or mac algorithm** |
| --- | --- |
| HS256 | HMAC using SHA-256 hash algorithm |
| HS384 | HMAC using SHA-384 hash algorithm |
| HS512 | HMAC using SHA-512 hash algorithm |
| RS256 | RSASSA using SHA-256 hash algorithm |
| RS384 | RSASSA using SHA-384 hash algorithm |
| RS512 | RSASSA using SHA-512 hash algorithm |
| ES256 | ECDSA using P-256 curve and SHA-256 hash algorithm |
| ES384 | ECDSA using P-384 curve and SHA-384 hash algorithm |
| ES512 | ECDSA using P-521 curve and SHA-512 hash algorithm |
| none | No digital signature or MAC value included |