Nodemailer

**Nodemailer** is a module for Node.js applications to allow easy as cake email sending.

### Nodemailer features

* A single module with **zero dependencies** – code is easily auditable, as there are no dark corners
* Heavy focus on **security**, no-one likes [RCE vulnerabilities](https://thehackernews.com/2017/01/phpmailer-swiftmailer-zendmail.html)
* **Unicode support** to use any characters, including emoji 💪
* **Windows support** – you can install it with [npm](https://www.npmjs.com/package/nodemailer) on Windows just like any other module, there are no compiled dependencies. Use it hassle free from Azure or from your Windows box
* Use **HTML content**, as well as **plain text** alternative
* Add [**Attachments**](https://nodemailer.com/message/attachments/) to messages
* [**Embedded**](https://nodemailer.com/message/embedded-images/) image attachments for HTML content – your design does not get blocked
* Secure email delivery using **TLS/STARTTLS**
* Different [**transport methods**](https://nodemailer.com/transports/) in addition to the built-in [**SMTP support**](https://nodemailer.com/smtp/)
* Sign messages with [**DKIM**](https://nodemailer.com/dkim/)
* Custom [**Plugin support**](https://nodemailer.com/plugins/) for manipulating messages
* Sane [**OAuth2**](https://nodemailer.com/smtp/oauth2/) authentication
* [**Proxies**](https://nodemailer.com/smtp/proxies/) for SMTP connections
* **ES6 code** – no more unintentional memory leaks, due to hoisted var’s

#### Requirements

* **Node.js v6+**. That’s it.

If you are able to run Node.js version 6 or newer, then you can use Nodemailer. There are no platform or resource specific requirements.

#### TL;DR

In short, what you need to do to send messages, would be the following:

1. Create a Nodemailer transporter using either [SMTP](https://nodemailer.com/smtp/) or [some other](https://nodemailer.com/transports/) transport mechanism
2. Set up [message options](https://nodemailer.com/message/) (who sends what to whom)
3. Deliver the message object using the **sendMail()** method of your previously created transporter

##### **Example**

This is a complete example to send an email with plain text and HTML body

'use strict';

const nodemailer = require('nodemailer');

// create reusable transporter object using the default SMTP transport

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true, // secure:true for port 465, secure:false for port 587

auth: {

user: 'username@example.com',

pass: 'userpass'

}

});

// setup email data with unicode symbols

let mailOptions = {

from: '"Fred Foo 👻" <foo@blurdybloop.com>', // sender address

to: 'bar@blurdybloop.com, baz@blurdybloop.com', // list of receivers

subject: 'Hello ✔', // Subject line

text: 'Hello world ?', // plain text body

html: '<b>Hello world ?</b>' // html body

};

// send mail with defined transport object

transporter.sendMail(mailOptions, (error, info) => {

if (error) {

return console.log(error);

}

console.log('Message %s sent: %s', info.messageId, info.response);

});

### Examples

* **Nodemailer AMQP example** is an example of using RabbitMQ to manage Nodemailer email messages. [Source](https://github.com/nodemailer/nodemailer-amqp-example).

# Usage

## Setting it up

Install Nodemailer from [npm](https://www.npmjs.com/package/nodemailer)

npm install nodemailer

To send emails you need a transporter object

let transporter = nodemailer.createTransport(transport[, defaults])

Where

* **transporter** is going to be an object that is able to send mail
* **transport** is the transport configuration object, connection url or a transport plugin instance
* **defaults** is an object that defines default values for mail options

You have to create the transporter object only once. If you already have a transporter object you can use it to send mail as much as you like.

### Send using SMTP

See the details about setting up a SMTP based transporter [here](https://nodemailer.com/smtp/).

### Send using a transport plugin

See the details about setting up a plugin based transporter [here](https://nodemailer.com/transports/).

## Sending mail

Once you have a transporter object you can send mail with it:

transporter.sendMail(data[, callback])

Where

* **data** defines the mail content (see [Message Configuration](https://nodemailer.com/message/) for possible options)
* **callback** is an optional callback function to run once the message is delivered or it failed
  + **err** is the error object if message failed
  + **info** includes the result, the exact format depends on the transport mechanism used
    - **info.messageId** most transports should return the final Message-Id value used with this property
    - **info.envelope** includes the envelope object for the message
    - **info.accepted** is an array returned by SMTP transports (includes recipient addresses that were accepted by the server)
    - **info.rejected** is an array returned by SMTP transports (includes recipient addresses that were rejected by the server)
    - **info.pending** is an array returned by Direct SMTP transport. Includes recipient addresses that were temporarily rejected together with the server response
    - **response** is a string returned by SMTP transports and includes the last SMTP response from the server

If the message includes several recipients then the message is considered sent if at least one recipient is accepted

If callback argument is not set then the method returns a Promise object. Nodemailer itself does not use Promises internally but it wraps the return into a Promise for convenience.

# SMTP? Say what?

You might wonder why you would need to set something up while in comparison PHP’s [mail](https://php.net/manual/en/function.mail.php) command works out of the box with no special configuration whatsoever. Just call **mail(…)** and you’re already sending mail. So what’s going on in Node.js?

The difference is in the software stack required for your application to work. While Node.js stack is thin, all you need for your app to work is the node binary, then PHP’s stack is fat. The server you’re running your PHP code on has several different components installed.

Firstly the PHP interpreter itself. Then there’s some kind of web server, most probably Apache or Nginx. Web server needs some way to interact with the PHP interpreter, so you have a (fast-)CGI process manager. There might be MySQL also running in the same host. Depending on the installation type you might even have imagemagick executables or other helpers lying around somewhere. And finally, you have the sendmail binary.

What PHP’s **mail()** call actually does is that it passes your mail data to sendmail’s stdin and thats it, no magic involved. sendmail does all the heavy lifting of queueing your message and trying to send it to the recipients’ MX mail server.

In fact you can achieve the exact same behavior in Nodemailer as well by using the [sendmail transport](https://nodemailer.com/transports/sendmail/). The difference being that in case of PHP the sendmail configuration resides in php.ini but in case of Node.js the sendmail configuration is part of Nodemailer setup.

Unfortunately this might not always work – as it was already said, the stack for Node.js is thin and this means that sendmail might not have been installed to the server your application is running on. This is why it is better to rely on an actual SMTP service that is always accessible.

# Using Gmail

Even though Gmail is the fastest way to get started with sending emails, it is by no means a preferable solution unless you are using OAuth2 authentication. Gmail expects the user to be an actual user not a robot so it runs a lot of heuristics for every login attempt and blocks anything that looks suspicious to defend the user from account hijacking attempts. For example you might run into trouble if your server is in another geographical location – everything works in your dev machine but messages are blocked in production.

Additionally Gmail has came up with the concept of [“Less Secure”](https://support.google.com/accounts/answer/6010255?hl=en) apps which is basically anyone who uses plain password to login to Gmail, so you might end up in a situation where one username can send mail (support for “less secure” apps is enabled) but other is blocked (support for “less secure” apps is disabled). You can configure your Gmail account to allow less secure apps [here](https://www.google.com/settings/security/lesssecureapps). When using this method make sure to also enable the required functionality by completing the [“Captcha Enable”](https://accounts.google.com/b/0/displayunlockcaptcha) challenge. Without this, less secure connections probably would not work.

If you are using 2FA you would have to create an [“Application Specific”](https://security.google.com/settings/security/apppasswords) password for Nodemailer to work.

Gmail also always sets authenticated username as the From: email address. So if you authenticate as foo@example.com and set bar@example.com as the from: address, then Gmail reverts this and replaces the sender with the authenticated user.

To prevent having login issues you should either use OAuth2 (see details [here](https://nodemailer.com/smtp/oauth2/)) or use another delivery provider and preferably a dedicated one. Usually these providers have free plans available that are comparable to the daily sending limits of Gmail. Gmail has a limit of 500 recipients a day (a message with one To and one Cc address counts as two messages since it has two recipients) for @gmail.com addresses and 2000 for Google Apps customers, larger SMTP providers usually offer about 200-300 recipients a day for free.

Delivering bulk mail

Here are some tips how to handle bulk mail, for example if you need to send 10 million messages at once.

1. **Use a dedicated delivery provider**. Do not use services that offer SMTP as a sideline or for free (that’s Gmail or the SMTP of your homepage hosting company) to send bulk mail – you’ll hit all the hard limits immediately or get labelled as spammer. Basically you get what you pay for and if you pay zero then your deliverability is near zero as well. Email might seem free but it is only free to a certain amount and that amount certainly does not include 10 million emails in a short period of time.
2. **Use a dedicated queue manager,** for example [RabbitMQ](https://www.rabbitmq.com/) for queueing the emails. Nodemailer creates a callback function with related scopes etc. for every message so it might be hard on memory if you pile up the data for 10 million messages at once. Better to take the data from a queue when there’s a free spot in the connection pool (previously sent message returns its callback).
3. **Use pooled SMTP** by setting *pool* option to *true* (assuming you always send using the same credentials). You do not want to have the overhead of creating a new connection and doing the SMTP handshake dance for every single email. Pooled connections make it possible to bring this overhead to a minimum.
4. **Set *maxMessages* option to *Infinity*** for the pooled SMTP transport. Dedicated SMTP providers happily accept all your emails as long you are paying for these, so no need to disconnect in the middle if everything is going smoothly. The default value is 100 which means that once a connection is used to send 100 messages it is removed from the pool and a new connection is created.
5. **Set *maxConnections* to whatever your system can handle.** There might be limits to this on the receiving side, so do not set it to *Infinity*, even 20 is probably much better than the default 5. A larger number means a larger amount of messages are sent in parallel.
6. **Use file paths not URLs for attachments.** If you are reading the same file from the disk several million times, the contents for the file probably get cached somewhere between your app and the physical hard disk, so you get your files back quicker (assuming you send the same attachment to all recipients). There is nothing like this for URLs – every new message makes a fresh HTTP fetch to receive the file from the server.
7. If the SMTP service accepts HTTP API as well you still might prefer SMTP and not the HTTP API as HTTP introduces additional overhead. You probably want to use HTTP over SMTP if the HTTP API is bulk aware – you send a message template and the list of 10 million recipients and the service compiles this information into emails itself, you can’t beat this with SMTP.

# Message configuration

The following are the possible fields of an email message:

### Commmon fields

* **from** - The email address of the sender. All email addresses can be plain ‘sender@server.com’ or formatted ’“Sender Name”[*sender@server.com*](mailto:sender@server.com)‘, see [Address object](https://nodemailer.com/message/addresses/) for details
* **to** - Comma separated list or an array of recipients email addresses that will appear on the To: field
* **cc** - Comma separated list or an array of recipients email addresses that will appear on the Cc: field
* **bcc** - Comma separated list or an array of recipients email addresses that will appear on the Bcc: field
* **subject** - The subject of the email
* **text** - The plaintext version of the message as an Unicode string, Buffer, Stream or an attachment-like object ({path: ‘/var/data/…’})
* **html** - The HTML version of the message as an Unicode string, Buffer, Stream or an attachment-like object ({path: ‘http://…‘})
* **attachments** - An array of attachment objects (see [Using attachments](https://nodemailer.com/message/attachments/) for details). Attachments can be used for [embedding images](https://nodemailer.com/message/embedded-images/) as well.

A large majority of emails sent look a lot like this, using only a few basic fields:

var message = {

from: 'sender@server.com',

to: 'receiver@sender.com',

subject: 'Message title',

text: 'Plaintext version of the message',

html: '<p>HTML version of the message</p>'

};

All text fields (email addresses, plaintext body, html body, attachment filenames) use UTF-8 as the encoding. Attachments are streamed as binary.

### More advanced fields

##### **Routing options**

* **sender** - An email address that will appear on the Sender: field (always prefer from if you’re not sure which one to use)
* **replyTo** - An email address that will appear on the Reply-To: field
* **inReplyTo** - The Message-ID this message is replying to
* **references** - Message-ID list (an array or space separated string)
* **envelope** - optional SMTP envelope, if auto generated envelope is not suitable (see [SMTP envelope](https://nodemailer.com/smtp/envelope/) for details)

##### **Content options**

* **attachDataUrls** – if true then convert data: images in the HTML content of this message to embedded attachments
* **watchHtml** - Apple Watch specific HTML version of the message
* **icalEvent** – iCalendar event to use as an alternative. See details [here](https://nodemailer.com/message/calendar-events/)
* **alternatives** - An array of alternative text contents (in addition to text and html parts) (see [Using alternative content](https://nodemailer.com/message/alternatives/) for details)
* **encoding** - identifies encoding for text/html strings (defaults to ‘utf-8’, other values are ‘hex’ and ‘base64’)
* **raw** - existing MIME message to use instead of generating a new one. See details [here](https://nodemailer.com/message/custom-source/)
* **textEncoding** - force content-transfer-encoding for text values (either quoted-printable or base64). By default the best option is detected (for lots of ascii use quoted-printable, otherwise base64)

##### **Header options**

* **priority** - Sets message importance headers, either **‘high’**, **‘normal’** (default) or **‘low’**.
* **headers** - An object or array of additional header fields (e.g. {“X-Key-Name”: “key value”} or [{key: “X-Key-Name”, value: “val1”}, {key: “X-Key-Name”, value: “val2”}]). Read more about custom headers [here](https://nodemailer.com/message/custom-headers/)
* **messageId** - optional Message-Id value, random value will be generated if not set
* **date** - optional Date value, current UTC string will be used if not set
* **list** - helper for setting List-\* headers (see more [here](https://nodemailer.com/message/list-headers/))

##### **Security options**

* **disableFileAccess** if true, then does not allow to use files as content. Use it when you want to use JSON data from untrusted source as the email. If an attachment or message node tries to fetch something from a file the sending returns an error. If this field is also set in the transport options, then the value in mail data is ignored
* **disableUrlAccess** if true, then does not allow to use Urls as content. If this field is also set in the transport options, then the value in mail data is ignored

var message = {

...,

headers: {

'My-Custom-Header': 'header value'

},

date: new Date('2000-01-01 00:00:00')

};

**Memory leak warning!** When using readable streams as content and sending fails then Nodemailer does not abort the already opened but not yet finished stream, you need to do this yourself. Nodemailer only closes the streams it has opened itself (eg. file paths, URLs)

var htmlstream = fs.createReadStream('content.html');

transport.sendMail({html: htmlstream}, function(err){

if(err){

// check if htmlstream is still open and close it to clean up

}

});

Attachments

**attachments** option in the message object that contains an array of attachment objects.

Attachment object consists of the following properties:

* **filename** - filename to be reported as the name of the attached file. Use of unicode is allowed.
* **content** - String, Buffer or a Stream contents for the attachment
* **path** - path to the file if you want to stream the file instead of including it (better for larger attachments)
* **href** – an URL to the file (data uris are allowed as well)
* **contentType** - optional content type for the attachment, if not set will be derived from the *filename* property
* **contentDisposition** - optional content disposition type for the attachment, defaults to ‘attachment’
* **cid** - optional content id for using inline images in HTML message source
* **encoding** - If set and *content* is string, then encodes the content to a Buffer using the specified encoding. Example values: *‘base64’*, *‘hex’*, *‘binary’* etc. Useful if you want to use binary attachments in a JSON formatted email object.
* **headers** - custom headers for the attachment node. Same usage as with message headers
* **raw** - is an optional special value that overrides entire contents of current mime node including mime headers. Useful if you want to prepare node contents yourself

Attachments can be added as many as you want.

**Example**

let message = {

...

attachments: [

{ // utf-8 string as an attachment

filename: 'text1.txt',

content: 'hello world!'

},

{ // binary buffer as an attachment

filename: 'text2.txt',

content: new Buffer('hello world!','utf-8')

},

{ // file on disk as an attachment

filename: 'text3.txt',

path: '/path/to/file.txt' // stream this file

},

{ // filename and content type is derived from path

path: '/path/to/file.txt'

},

{ // stream as an attachment

filename: 'text4.txt',

content: fs.createReadStream('file.txt')

},

{ // define custom content type for the attachment

filename: 'text.bin',

content: 'hello world!',

contentType: 'text/plain'

},

{ // use URL as an attachment

filename: 'license.txt',

path: 'https://raw.github.com/nodemailer/nodemailer/master/LICENSE'

},

{ // encoded string as an attachment

filename: 'text1.txt',

content: 'aGVsbG8gd29ybGQh',

encoding: 'base64'

},

{ // data uri as an attachment

path: 'data:text/plain;base64,aGVsbG8gd29ybGQ='

},

{

// use pregenerated MIME node

raw: 'Content-Type: text/plain\r\n' +

'Content-Disposition: attachment;\r\n' +

'\r\n' +

'Hello world!'

}

]

}

# Alternatives

In addition to text and HTML, any kind of data can be inserted as an alternative content of the main body - for example a word processing document with the same text as in the HTML field. It is the job of the email client to select and show the best fitting alternative to the reader. Usually this field is used for calendar events and such.

If you want to use a calendar event as the alternative, the consider using the **icalEvent** option instead. See details [here](https://nodemailer.com/message/calendar-events/).

Alternative objects use the same options as [attachment objects](https://nodemailer.com/attachments/). The difference between an attachment and an alternative is the fact that attachments are placed into multipart/mixed or multipart/related parts of the message white alternatives are placed into multipart/alternative part.

**Usage example:**

let message = {

...

html: '<b>Hello world!</b>',

alternatives: [

{

contentType: 'text/x-web-markdown',

content: '\*\*Hello world!\*\*'

}

]

}

Alternatives can be added as many as you want.

Address object

All email addresses can be **plain** email addresses

'foobar@blurdybloop.com'

* or with **formatted name** (includes unicode support)

'Ноде Майлер <foobar@blurdybloop.com>'

Notice that all address fields (even *from:*) are comma separated lists, so if you want to use a comma (or any other special symbol) in the name part, make sure you enclose the name in double quotes like this: '"Майлер, Ноде" <foobar@blurdybloop.com>'

* or as an **address object** (in this case you do not need to worry about the formatting, no need to use quotes etc.)

{

name: 'Майлер, Ноде',

address: 'foobar@blurdybloop.com'

}

All address fields accept values as a comma separated list of emails or an array of emails or an array of comma separated list of emails or address objects - use it as you like. Formatting can be mixed.

...,

to: 'foobar@blurdybloop.com, "Ноде Майлер" <bar@blurdybloop.com>, "Name, User" <baz@blurdybloop.com>',

cc: [

'foobar@blurdybloop.com',

'"Ноде Майлер" <bar@blurdybloop.com>,

"Name, User" <baz@blurdybloop.com>'

],

bcc: [

'foobar@blurdybloop.com',

{

name: 'Майлер, Ноде',

address: 'foobar@blurdybloop.com'

}

]

...

You can even use unicode domains, these are automatically converted to punycode

'"Unicode Domain" <info@müriaad-polüteism.info>'

# Calendar events

Calendar events are tricky because different email clients handle these differently. Nodemailer uses the same style as Gmail for attaching calendar files which should ensure maximum compatibility. If you want to attach a calendar event to your email then you can use the message option **icalEvent**:

* **icalEvent** – an object to define calendar event
  + **method** – optional method, case insensitive, defaults to ‘publish’. Other possible values would be ‘request’, ‘reply’, ‘cancel’ or any other valid calendar method listed in [RFC5546](https://tools.ietf.org/html/rfc5546#section-1.4). This should match the **METHOD:** value in calendar event file.
  + **filename** – optional filename, defaults to ‘invite.ics’
  + **content** – is the event file, it can be a string, a buffer, a stream
  + **path** – is an alternative for content to load the calendar data from a file
  + **href** – is an alternative for content to load the calendar data from an URL
  + **encoding** – defines optional content encoding, eg. ‘base64’ or ‘hex’. This only applies if the content is a string. By default an unicode string is assumed.

You can use modules like [ical-generator](https://www.npmjs.com/package/ical-generator) to generate the actual calendar file content, Nodemailer acts as a transport layer only and does not generate the event file structure.

In general it is not a good idea to add additional attachments to calendar messages as it might mess up the behavior of some email clients. Try to keep it only to **text**, **html** and **icalEvent** without any additional **alternatives** or **attachments**

# Examples

## 1. Send a REQUEST event as a string

let content = 'BEGIN:VCALENDAR\r\nPRODID:-//ACME/DesktopCalendar//EN\r\nMETHOD:REQUEST\r\n...';

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Appointment',

text: 'Please see the attached appointment',

icalEvent: {

filename: 'invitation.ics',

method: 'request',

content: content

}

};

## 2. Send a PUBLISH event from a file

Event data is loaded from the provided file and attached to the message.

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Appointment',

text: 'Please see the attached appointment',

icalEvent: {

method: 'PUBLISH',

path: '/path/to/file'

}

};

## 3. Send a CANCEL event from an URL

Event data is downloaded from the provided URL and attached to the message as regular calendar file.

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Appointment',

text: 'Please see the attached appointment',

icalEvent: {

method: 'CANCEL',

href: 'http://www.example.com/events?event=123'

}

};

# Embedded images

Attachments can be used as embedded images in the HTML body. To use this feature, you need to set additional property of the attachment - **cid** (unique identifier of the file) which is a reference to the attachment file. The same **cid** value must be used as the image URL in HTML (using **cid:** as the URL protocol, see example below).

**NB!** the cid value should be as unique as possible!

#### Example

let message = {

...

html: 'Embedded image: <img src="cid:unique@nodemailer.com"/>',

attachments: [{

filename: 'image.png',

path: '/path/to/file',

cid: 'unique@nodemailer.com' //same cid value as in the html img src

}]

}

# List headers

Nodemailer includes a helper for setting more complex List-\* headers with ease. You can use this by providing message option **list**. It’s an object where key names are converted into list headers. List key help becomes List-Help header etc.

**General rules**

* If the value is a string, it is treated as an URL
* If you want to provide an optional comment, use {url:‘url’, comment: ‘comment’} object
* If you want to have multiple header rows for the same List-\* key, use an array as the value for this key
* If you want to have multiple URLs for single List-\* header row, use an array inside an array
* List-\* headers are treated as pregenerated values, this means that lines are not folded and strings are not encoded. Use only ascii characters and be prepared for longer header lines

### Examples

#### 1. Setup different List-\* headers

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'List Message',

text: 'I hope no-one unsubscribes from this list!',

list: {

// List-Help: <mailto:admin@example.com?subject=help>

help: 'admin@example.com?subject=help',

// List-Unsubscribe: <http://example.com> (Comment)

unsubscribe: {

url: 'http://example.com',

comment: 'Comment'

},

// List-Subscribe: <mailto:admin@example.com?subject=subscribe>

// List-Subscribe: <http://example.com> (Subscribe)

subscribe: [

'admin@example.com?subject=subscribe',

{

url: 'http://example.com',

comment: 'Subscribe'

}

],

// List-Post: <http://example.com/post>, <mailto:admin@example.com?subject=post> (Post)

post: [

[

'http://example.com/post',

{

url: 'admin@example.com?subject=post',

comment: 'Post'

}

]

]

}

};

# Custom headers

Most messages do not need any kind of tampering with the headers. If you do need to add custom headers either to the message or to an attachment/alternative, you can add these values with the **headers** option. Values are processed automatically, non-ascii strings are encoded as mime-words and long lines are folded.

* **headers** – is an object of key-value pairs, where key names are converted into message header keys

### Examples

#### 1. Set custom headers

let message = {

...,

headers: {

'x-my-key': 'header value',

'x-another-key': 'another value'

}

}

// Becomes:

// X-My-Key: header value

// X-Another-Key: another value

#### 2. Multiple rows with the same key

The same header key can be used multiple times if the header value is an Array

let message = {

...,

headers: {

'x-my-key': [

'value for row 1',

'value for row 2',

'value for row 3'

]

}

}

// X-My-Key: value for row 1

// X-My-Key: value for row 2

// X-My-Key: value for row 3

#### 3. Prepared headers

Normally all headers are encoded and folded to meet the requirement of having plain-ASCII messages with lines no longer than 78 bytes. Sometimes it is preferable to not modify header values and pass these as provided. This can be achieved with the **prepared** option:

let message = {

...,

headers: {

'x-processed': 'a really long header or value with non-ascii characters 👮',

'x-unprocessed': {

prepared: true,

value: 'a really long header or value with non-ascii characters 👮'

}

}

}

// X-Processed: a really long header or value with non-ascii characters

// =?UTF-8?Q?=F0=9F=91=AE?=

// X-Unprocessed: a really long header or value with

# Custom source

If you want to use your own custom generated RFC822 formatted message source, instead of letting Nodemailer to generate it from the message options, use option **raw**. This can be used both for the entire message or alternatively per-attachment or per-alternative.

Don’t forget to set the **envelope** option when using **raw** as the message source

### Examples

#### 1. Use string as a message body

This example loads the entire message source from a string value. You don’t have to worry about proper newlines, these are handled by Nodemailer.

let message = {

envelope: {

from: 'sender@example.com',

to: ['recipient@example.com']

},

raw: `From: sender@example.com

To: recipient@example.com

Subject: test message

Hello world!`

};

#### 2. Set EML file as message body

This example loads the entire message source from a file

let message = {

envelope: {

from: 'sender@example.com',

to: ['recipient@example.com']

},

raw: {

path: '/path/to/message.eml'

}

};

#### 3. Set string as attachment body

When using **raw** for attachments then you need to provide all content headers youself, Nodemailer does not process it in any way (besides newline processing), it is inserted into the MIME tree as is.

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Custom attachment',

attachments: [{

raw: `Content-Type: text/plain

Content-Disposition: attachment

Attached text file`}]

};

# SMTP transport

SMTP is the main transport in Nodemailer for delivering messages. SMTP is also the protocol used between different email hosts, so its truly universal. Almost every email delivery provider supports SMTP based sending, even if they mainly push their API based sending. APIs might have more features but using these also means vendor lock-in while in case of SMTP you only need to change the configuration options to replace one provider with another and you’re good to go.

let transporter = nodemailer.createTransport(options[, defaults])

Where

* **options** – is an object that defines connection data (see below for details)
* **defaults** – is an object that is going to be merged into every message object. This allows you to specify shared options, for example to set the same from address for every message

Alternatively you could use a connection url instead of an object for the options. Use ‘smtp:’ or ‘smtps:’ as the protocol in the url.

let poolConfig = 'smtps://username:password@smtp.example.com/?pool=true';

##### **General options**

* **port** – is the port to connect to (defaults to 587 is secure is false or 465 if true)
* **host** – is the hostname or IP address to connect to (defaults to ‘localhost’)
* **auth** – defines authentication data (see [authentication](https://nodemailer.com/smtp/#authentication) section below)
* **authMethod** – defines preferred authentication method, defaults to ‘PLAIN’

##### **TLS options**

* **secure** – if true the connection will use TLS when connecting to server. If false (the default) then TLS is used if server supports the STARTTLS extension. In most cases set this value to true if you are connecting to port 465. For port 587 or 25 keep it false
* **tls** – defines additional [node.js TLSSocket options](https://nodejs.org/api/tls.html#tls_class_tls_tlssocket) to be passed to the socket constructor, eg. {rejectUnauthorized: true}.
* **ignoreTLS** – if this is true and secure is false then TLS is not used even if the server supports STARTTLS extension
* **requireTLS** – if this is true and secure is false then Nodemailer tries to use STARTTLS even if the server does not advertise support for it. If the connection can not be encrypted then message is not sent

Setting **secure** to **false** does not mean that you would not use an encrypted connection. Most SMTP servers allow connection upgrade via [STARTTLS](https://tools.ietf.org/html/rfc3207#section-2) command but to use this you have to connect using plaintext first

##### **Connection options**

* **name** – optional hostname of the client, used for identifying to the server, defaults to hostname of the machine
* **localAddress** – is the local interface to bind to for network connections
* **connectionTimeout** – how many milliseconds to wait for the connection to establish
* **greetingTimeout** – how many milliseconds to wait for the greeting after connection is established
* **socketTimeout** – how many milliseconds of inactivity to allow

##### **Debug options**

* **logger** – optional [bunyan](https://github.com/trentm/node-bunyan) compatible logger instance. If set to true then logs to console. If value is not set or is false then nothing is logged
* **debug** – if set to true, then logs SMTP traffic, otherwise logs only transaction events

##### **Security options**

* **disableFileAccess** – if true, then does not allow to use files as content. Use it when you want to use JSON data from untrusted source as the email. If an attachment or message node tries to fetch something from a file the sending returns an error
* **disableUrlAccess** – if true, then does not allow to use Urls as content

##### **Pooling options**

* **pool** – see [Pooled SMTP](https://nodemailer.com/smtp/pooled/) for details about connection pooling

##### **Proxy options**

* **proxy** – all SMTP based transports allow to use proxies for making TCP connections to servers. Read about proxy support in Nodemailer from [here](https://nodemailer.com/smtp/proxies/)

### Examples

#### 1. Single connection

This example would connect to a SMTP server separately for every single message

let smtpConfig = {

host: 'smtp.example.com',

port: 587,

secure: false, // upgrade later with STARTTLS

auth: {

user: 'username',

pass: 'password'

}

};

#### 2. Single connection

This example would set up pooled connections against a SMTP server on port 465

let poolConfig = {

pool: true,

host: 'smtp.example.com',

port: 465,

secure: true, // use TLS

auth: {

user: 'username',

pass: 'password'

}

};

#### 3. Allow self-signed certificates

This config would open a connection to TLS server with self-signed or invalid TLS certificate

let selfSignedConfig = {

host: 'my.smtp.host',

port: 465,

secure: true, // use TLS

auth: {

user: 'username',

pass: 'pass'

},

tls: {

// do not fail on invalid certs

rejectUnauthorized: false

}

};

## Authentication

If authentication data is not present, the connection is considered authenticated from the start. Otherwise you would need to provide the authentication options object.

* **auth** is the authentication object
  + **type** indicates the authetication type, defaults to ‘login’, other option is ‘oauth2’
  + **user** is the username
  + **pass** is the password for the user if normal login is used

For authenticating using OAuth2 instead of normal auth, see OAuth2 options for the **auth** object [here](https://nodemailer.com/smtp/oauth2/).

## Verify SMTP connection configuration

You can verify your SMTP configuration with **verify(callback)** call (also works as a Promise). If it returns an error, then something is not correct, otherwise the server is ready to accept messages.

// verify connection configuration

transporter.verify(function(error, success) {

if (error) {

console.log(error);

} else {

console.log('Server is ready to take our messages');

}

});

Be aware though that this call only tests connection and authentication but it does not check if the service allows you to use a specific envelope From address or not.

SMTP envelope

SMTP envelope is usually auto generated from **from**, **to**, **cc** and **bcc** fields in the message object but if for some reason you want to specify it yourself (custom envelopes are usually used for VERP addresses), you can do it with the **envelope** property in the message object.

* **envelope** – is an object with the following address params that behave just like with regular mail options. You can also use the regular address format, unicode domains etc.
  + **from** – the first address gets used as MAIL FROM address in SMTP
  + **to** – addresses from this value get added to RCPT TO list
  + **cc** – addresses from this value get added to RCPT TO list
  + **bcc** – addresses from this value get added to RCPT TO list

let message = {

...,

from: 'mailer@nodemailer.com', // listed in rfc822 message header

to: 'daemon@nodemailer.com', // listed in rfc822 message header

envelope: {

from: 'Daemon <deamon@nodemailer.com>', // used as MAIL FROM: address for SMTP

to: 'mailer@nodemailer.com, Mailer <mailer2@nodemailer.com>' // used as RCPT TO: address for SMTP

}

}

The envelope object returned by **sendMail()** includes just **from** (address string) and **to** (an array of address strings) fields as all addresses from **to**, **cc** and **bcc** get merged into **to** when sending.

# Pooled SMTP

If pooling is used then Nodemailer keeps a fixed amount of connections open and sends the next message once a connection becomes available. It is mostly useful when you have a large number of messages that you want to send in batches or your provider allows you to only use a small amount of parallel connections.

To use pooled connections use the following options in transport configuration

* **pool** – set to true to use pooled connections (defaults to false) instead of creating a new connection for every email
* **maxConnections** – is the count of maximum simultaneous connections to make against the SMTP server (defaults to 5)
* **maxMessages** – limits the message count to be sent using a single connection (defaults to 100). After maxMessages is reached the connection is dropped and a new one is created for the following messages
* **rateDelta** – defines the time measuring period in milliseconds (defaults to 1000, ie. to 1 second) for rate limiting
* **rateLimit** – limits the message count to be sent in **rateDelta** time. Once rateLimit is reached, sending is paused until the end of the measuring period. This limit is shared between connections, so if one connection uses up the limit, then other connections are paused as well. If rateLimit is not set then sending rate is not limited

### Methods

#### transporter.isIdle()

Returns true if there are available connection slots

#### transporter.close()

If transporter uses pooling then connections are kept open even if there is nothing to be sent. To close all pending connections you can use the close() method

let transporter = nodemailer.createTransport({pool: true,...});

// ...

transporter.close();

### Events

#### Event:‘idle’

Emitted by the transporter object if connection pool has free connection slots. Check if a connection is still available with isIdle() method (returns true if a connection is still available). This allows to create push-like senders where messages are not queued into memory in a Node.js process but pushed and loaded through an external queue.

let messages = [...'list of messages'];

transporter.on('idle', function(){

// send next message from the pending queue

while (transporter.isIdle() && messages.length) {

transporter.sendMail (messages.shift());

}

});

# OAuth2

OAuth2 allows your application to store and use authentication tokens instead of actual login credentials. This is great for security as tokens or valid only for specific actions and can be easily revoked thus, once stolen, can’t to as much harm as actual account credentials. OAuth2 authentication in Nodemailer is mostly used with Gmail and G Suite (née Google Apps) even though there are other providers that support it as well.

Access Tokens needed for OAuth2 authentication are short lived so these need to be regenerated from time to time. Nodemailer is able to use both [3LO](https://developers.google.com/identity/protocols/OAuth2) and [2LO](https://developers.google.com/api-client-library/php/auth/service-accounts) to automatically regenerate the tokens but you can also handle all token specific yourself.

1. [Normal OAuth2 authentication](https://nodemailer.com/smtp/oauth2/#oauth-3lo)
2. [Authenticating using Service Accounts](https://nodemailer.com/smtp/oauth2/#oauth-2lo)
3. [Using custom token handling](https://nodemailer.com/smtp/oauth2/#custom-handling)
4. [Token update notifications](https://nodemailer.com/smtp/oauth2/#update-notification)
5. [Examples](https://nodemailer.com/smtp/oauth2/#examples)
6. [Troubleshooting](https://nodemailer.com/smtp/oauth2/#troubleshooting)

Nodemailer requires an **Access Token** to perform authentication. 3-legged and 2-legged OAuth2 mechanisms are different ways to produce such tokens but in the end it does not matter how a token was exactly generated, as long as it is valid.

### 3-legged OAuth2 authentication

This is the “normal” way of obtaining access tokens. Your application requests permissions from the client and gets a refresh token in return that can be used to generate new access tokens.

* **auth** – is the authentication object
  + **type** – indicates authentication type, set it to ‘OAuth2’
  + **user** – user email address (required)
  + **clientId** – is the registered client id of the application
  + **clientSecret** – is the registered client secret of the application
  + **refreshToken** – is an optional refresh token. If it is provided then Nodemailer tries to generate a new access token if existing one expires or fails
  + **accessToken** – is the access token for the user. Required only if refreshToken is not available and there is no token refresh callback specified
  + **expires** – is an optional expiration time for the current accessToken
  + **accessUrl** – is an optional HTTP endpoint for requesting new access tokens. This value defaults to Gmail

Normal SMTP transport (ie. not the pooled version) has a convenience method of using separate authentication for every message. This allows you to set up a transport with just clientId and clientSecret values and provide accessToken and refreshToken with the message options. See [example 5](https://nodemailer.com/smtp/oauth2/#example-5).

### 2LO authentication (service accounts)

Nodemailer also allows you to use [service accounts](https://developers.google.com/identity/protocols/OAuth2ServiceAccount) to generate access tokens. In this case the required auth options are a bit different from 3LO auth.

* **auth** – is the authentication object
  + **type** – indicates authentication type, set it to ‘OAuth2’
  + **user** – user email address you want to send mail as (required)
  + **serviceClient** – service client id (required), you can find it from the “client\_id” field in the service key file
  + **privateKey** – is the private key contents, you can find it from the “private\_key” field in the service key file

### Using custom token handling

If you do not want Nodemailer to create new access tokens then you can provide a custom token generation callback that is called every time a new token is needed for an user.

The registered function gets the following arguments:

* **user** – is the user email address
* **renew** – if true then previous access token either expired or it was not accepted by the SMTP server, in this case you should generate a new value
* **callback** with arguments (err, accessToken) – is the callback function to run once you have generated a new access token

transporter.set('oauth2\_provision\_cb', (user, renew, callback)=>{

let accessToken = userTokens[user];

if(!accessToken){

return callback(new Error('Unknown user'));

}else{

return callback(null, accessToken);

}

});

### Token update notifications

If you use refreshToken or service keys to generate new tokens from Nodemailer when accessToken is not present or expired then you can listen for the token updates by registering a ‘token’ event handler for the transporter object.

transporter.on('token', token => {

console.log('A new access token was generated');

console.log('User: %s', token.user);

console.log('Access Token: %s', token.accessToken);

console.log('Expires: %s', new Date(token.expires));

});

### Examples

#### 1. Authenticate using existing token

Use an existing Access Token. If the token is not accepted then message is not sent as there is no way to generate a new token.

let transporter = nodemailer.createTransport({

host: 'smtp.gmail.com',

port: 465,

secure: true,

auth: {

type: 'OAuth2',

user: 'user@example.com',

accessToken: 'ya29.Xx\_XX0xxxxx-xX0X0XxXXxXxXXXxX0x'

}

});

#### 2. Custom handler

This example requests a new accessToken value from a custom OAuth2 handler. Nodemailer does not attempt to generate the token by itself.

let transporter = nodemailer.createTransport({

host: 'smtp.gmail.com',

port: 465,

secure: true,

auth: {

type: 'OAuth2',

user: 'user@example.com'

}

});

transporter.set('oauth2\_provision\_cb', (user, renew, callback)=>{

let accessToken = userTokens[user];

if(!accessToken){

return callback(new Error('Unknown user'));

}else{

return callback(null, accessToken);

}

});

#### 3. Set up 3LO authentication

This example uses an existing Access Token. If the token is not accepted or current time is past the expires value, then refreshToken is used to automatically generate a new accessToken

let transporter = nodemailer.createTransport({

host: 'smtp.gmail.com',

port: 465,

secure: true,

auth: {

type: 'OAuth2',

user: 'user@example.com',

clientId: '000000000000-xxx0.apps.googleusercontent.com',

clientSecret: 'XxxxxXXxX0xxxxxxxx0XXxX0',

refreshToken: '1/XXxXxsss-xxxXXXXXxXxx0XXXxxXXx0x00xxx',

accessToken: 'ya29.Xx\_XX0xxxxx-xX0X0XxXXxXxXXXxX0x',

expires: 1484314697598

}

});

#### 4. Set up 2LO authentication

This example uses an existing Access Token. If the token is not accepted or current time is past the expires value, then a new accessToken value is generated using provided service account.

let transporter = nodemailer.createTransport({

host: 'smtp.gmail.com',

port: 465,

secure: true,

auth: {

type: 'OAuth2',

user: 'user@example.com',

serviceClient: '113600000000000000000',

privateKey: '-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBg...',

accessToken: 'ya29.Xx\_XX0xxxxx-xX0X0XxXXxXxXXXxX0x',

expires: 1484314697598

}

});

#### 5. Provide authentication details with message options

This example demonstrates how to authenticate every message separately. This is mostly useful if you provide an email application that sends mail for multiple users. Instead of creating a new transporter for every message, create it just once and provide dynamic details with the message options.

Per-message specific authentication does not work in pooled mode

let transporter = nodemailer.createTransport({

host: 'smtp.gmail.com',

port: 465,

secure: true,

auth: {

type: 'OAuth2',

clientId: '000000000000-xxx.apps.googleusercontent.com',

clientSecret: 'XxxxxXXxX0xxxxxxxx0XXxX0'

}

});

transporter.sendMail({

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Message',

text: 'I hope this message gets through!',

auth: {

user: 'user@example.com',

refreshToken: '1/XXxXxsss-xxxXXXXXxXxx0XXXxxXXx0x00xxx',

accessToken: 'ya29.Xx\_XX0xxxxx-xX0X0XxXXxXxXXXxX0x',

expires: 1484314697598

}

});

Or alternatively you can do the same with your own OAuth2 handler.

let transporter = nodemailer.createTransport({

host: 'smtp.gmail.com',

port: 465,

secure: true,

auth: {

type: 'OAuth2'

}

});

transporter.set('oauth2\_provision\_cb', (user, renew, callback) => {

let accessToken = userTokens[user];

if(!accessToken){

return callback(new Error('Unknown user'));

}else{

return callback(null, accessToken);

}

});

transporter.sendMail({

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Message',

text: 'I hope this message gets through!',

auth: {

user: 'user@example.com'

}

});

### Troubleshooting

* The correct OAuth2 scope for Gmail SMTP is https://mail.google.com/, make sure your client has this scope set when requesting permissions for an user
* Make sure that Gmail API access is enabled for your Client ID. To do this, search for the Gmail API in [Google API Manager](https://console.developers.google.com/) and click on “enable”

# Proxy support

Nodemailer is able to use proxies for connecting to SMTP servers. HTTP proxy support is built in, Socks proxy support can be enabled by providing [socks](https://www.npmjs.com/package/socks) module to Nodemailer, other proxies need custom handling.

To enable proxying, define a **proxy** option for the transporter.

* **proxy** – is a proxy URL

### Examples

#### 1. Using HTTP proxy

Set HTTP proxy url for the proxy option. That’s it, everything required to handle it is built into Nodemailer.

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

proxy: 'http://proxy-host:1234'

});

Or if you want to use some environment defined variable like http\_proxy:

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

proxy: process.env.http\_proxy

});

Make sure that your HTTP proxy supports CONNECT protocol and allows connecting to the SMTP port you want to use.

#### 2. Using Socks proxy

Set Socks proxy url for the **proxy** option. Additionally you need to provide the [socks](https://www.npmjs.com/package/socks) module for the transporter as it is not bundled with Nodemailer.

Possible protocol values for the SOCKS proxy:

* ‘socks4:’ or ‘socks4a:’ for a SOCKS4 proxy
* ‘socks5:’ or ‘socks:’ for a SOCKS5 proxy

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

proxy: 'socks5://socks-host:1234'

});

// enable support for socks URLs

transporter.set('proxy\_socks\_module', require('socks'));

##### **Testing Socks proxies**

For testing you can use ssh to create a SOCKS5 proxy. The following command connects to your remote server and sets up a proxy on port 1080 that routes connections through that server.

ssh -N -D 0.0.0.0:1080 username@remote.host`

**proxy** url for that server would be socks5://localhost:1080

#### 3. Using a custom proxy handler

Additionally you can create your own proxy handler. To do this you would need to register a protocol handler callback with the name proxy\_handler\_{protocol} where {protocol} would be the protocol from proxy URL. If the URL looks like ‘yyy://localhost’ then you would need to set callback for proxy\_handler\_yyy.

transporter.set('proxy\_handler\_myproxy', handler)

Where

* **handler** is the function to run to create a proxied socket. It gets the following arguments:
  + **proxy** is the proxy url in a parsed form
  + **options** is transport configuration object
  + **callback** is the function to return the socket

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

proxy: 'myproxy://localhost:1234'

});

// enable support for socks URLs

transporter.set('proxy\_handler\_myproxy', (proxy, options, callback)=>{

console.log('Proxy host=% port=%', proxy.hostname, proxy.port);

let socket = require('net').connect(options.port, options.host, () => {

callback(null, {

connection: socket

});

});

});

If your proxy uses an encrypted connection then you can mark the proxied socket to be already secure. This prevents Nodemailer from upgrading the provided connection using TLS.

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

proxy: 'myproxys://localhost:1234'

});

// enable support for socks URLs

transporter.set('proxy\_handler\_myproxys', (proxy, options, callback)=>{

console.log('Proxy host=% port=%', proxy.hostname, proxy.port);

let socket = require('tls').connect(options.port, options.host, () => {

callback(null, {

connection: socket,

secured: true

});

});

});

# Delivery status notifications

If your delivery service supports it (not all SMTP servers have DSN extension enabled), then you can use Delivery status notifications (DSN) with Nodemailer as defined in [RFC3461](https://tools.ietf.org/html/rfc3461).

To set up a DSN call, add a dsn property to message data

* **dsn** – optional object to define DSN options
  + **id** – is the envelope identifier that would be included in the response (ENVID)
  + **return** – is either ‘headers’ or ‘full’. It specifies if only headers or the entire body of the message should be included in the response (RET)
  + **notify** – is either a string or an array of strings that define the conditions under which a DSN response should be sent. Possible values are ‘never’, ‘success’, ‘failure’ and ‘delay’. The condition ‘never’ can only appear on its own, other values can be grouped together into an array (NOTIFY)
  + **recipient** – is the email address the DSN should be sent (ORCPT)

Non-xtext strings are encoded automatically.

### Examples

#### 1. Request DSN for delivered messages

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Message',

text: 'I hope this message gets read!',

dsn: {

id: 'some random message specific id',

return: 'headers',

notify: 'success',

recipient: 'sender@example.com'

}

};

#### 2. Request DSN for undelivered and delayed messages

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Message',

text: 'I hope this message gets read!',

dsn: {

id: 'some random message specific id',

return: 'headers',

notify: ['failure', 'delay'],

recipient: 'sender@example.com'

}

};

# Plugins

Nodemailer can be extended with plugins. In most cases there are 3 different kind of plugins.

1. Plugins that operate on the mail object (for example extend it or change existing values). This is the pre-processing step
2. Plugins that operate on mail stream (for example to calculate message signatures). This is the processing step
3. Plugins that operate as transports (for example to send the message using an API of some delivery provider). This is the sending step

### Available plugins

These are some existing public plugins for Nodemailer

* [express-handlebars](https://github.com/yads/nodemailer-express-handlebars) – this plugin uses the express-handlebars view engine to generate html emails
* [inline-base64](https://github.com/mixmaxhq/nodemailer-plugin-inline-base64) – This plugin will convert base64-encoded images in your nodemailer email to be inline (“CID-referenced”) attachments within the email
* [html-to-text](https://github.com/andris9/nodemailer-html-to-text) – The plugin checks if there is no text option specified and populates it based on the html value

For a more extensive list [search for nodemailer](https://www.npmjs.com/search?q=nodemailer) in npm.

# DKIM signing

Nodemailer is able to sign all messages with DKIM keys. This means calculating a signature for the message and adding it as an additional header (or headers, if you use multiple keys) to the message.

The drawback on DKIM signing is that Nodemailer needs to cache the entire message before it can be sent, unlike normal sending where message output is streamed to SMTP as it is created and nothing needs to be cached. For small messages it does not matter, for larger messages Nodemailer offers an option to cache messages not in memory but on disk. In this case Nodemailer starts buffering the message in memory and if the message size reaches a certain treshold, it gets directed to a file on disk. Once signature is calculated and sent to SMTP, Nodemailer streams the cached message from disk to SMTP.

In general DKIM signing should be fast and effective.

### Setting it up

DKIM signing can be set on the transport level (all messages get signed with the same keys) and also on the message level (provide different keys for every message). If both are set, then message level DKIM configuration is preferred.

In both cases you need to provide a dkim object with the following structure

* **dkim** is an object with DKIM options
  + **domainName** – is the domain name to use in the signature
  + **keySelector** – is the DKIM key selector
  + **privateKey** – is the private key for the selector in PEM format
  + **keys** – is an optional array of key objects (domainName, keySelector, privateKey) if you want to add more than one signature to the message. If this value is set then the default key values are ignored
  + **cacheDir** – optional location for cached messages. If not set then caching is not used.
  + **cacheTreshold** – optional size in bytes, if message is larger than this treshold it gets cached to disk (assuming cacheDir is set and writable). Defaults to 131072 (128 kB).
  + **hashAlgo** – optional algorithm for the body hash, defaults to ‘sha256’
  + **headerFieldNames** – an optional colon separated list of header keys to sign (eg. message-id:date:from:to...')
  + **skipFields** – optional colon separated list of header keys not to sign. This is useful if you want to sign all the relevant keys but your provider changes some values, ie Message-ID and Date. In this case you should use 'message-id:date' to prevent signing these values.

### Examples

#### 1. Sign all messages

Assumes that there is a public key available for 2017.\_domainkey.example.com. You can test if the key exists or not with the dig tool like this

dig TXT 2017.\_domainkey.example.com

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

dkim: {

domainName: 'example.com',

keySelector: '2017',

privateKey: '-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBg...'

}

});

#### 2. Sign all messages with multiple keys

Assumes that there is a public keys available for 2017.\_domainkey.example.com and 2016.\_domainkey.example.com

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

dkim: {

keys: [

{

domainName: 'example.com',

keySelector: '2017',

privateKey: '-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBg...'

},

{

domainName: 'example.com',

keySelector: '2016',

privateKey: '-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBg...'

}

]

cacheDir: false

}

});

#### 3. Sign a specific message

Do not sign by default. Provide DKIM key values separately for every message.

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true

});

let message = {

from: 'sender@example.com',

to: 'recipient@example.com',

subject: 'Message',

text: 'I hope this message gets read!',

dkim: {

domainName: 'example.com',

keySelector: '2017',

privateKey: '-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBg...'

}

};

#### 4. Cache large messages for signing

Messages larger than 100kB are cached to disk

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

dkim: {

domainName: 'example.com',

keySelector: '2017',

privateKey: '-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBg...',

cacheDir: '/tmp',

cacheTreshold: 100 \* 1024

}

});

#### 5. Do not sign specific header keys

This is needed when sending mail through SES that has its own Message-ID and Date system.

let transporter = nodemailer.createTransport({

host: 'smtp.example.com',

port: 465,

secure: true,

dkim: {

domainName: 'example.com',

keySelector: '2017',

privateKey: '-----BEGIN PRIVATE KEY-----\nMIIEvgIBADANBg...',

skipFields: 'message-id:date'

}

});