nodemailer / **nodemailer**

Code Issues 11 Pull requests 0 Projects 0 Pulse Graphs

Send e-mails with Node.JS – easy as cake! http://nodemailer.com/

719 commits 8 branches 169 releases 77 contributors MIT

Branch: master New pull request Find file Clone or download

andris9 committed on GitHub Merge pull request #676 from sadika9/patch-1 ... Latest commit ee795f7 5 days ago

assets	Added Nodemailer logo	3 years ago
examples	v2.6.0	a month ago
lib	v2.6.0	a month ago
test	v2.6.0	a month ago
.eslintrc.js	v2.2.0-rc.8	8 months ago
.gitignore	Updated .gitignore	3 years ago
.npmignore	Bumped version to v1.10.0	11 months ago
.travis.yml	Bumped deps	3 months ago
CHANGELOG.md	v2.6.0	a month ago
Gruntfile.js	Bumped deps	3 months ago
ISSUE_TEMPLATE.md	v2.2.0-rc.11	8 months ago
LICENSE	Bumped version to v2.0.0-beta.2	10 months ago
README.md	Fix typo	5 days ago
package.json	updated package.json	26 days ago

README.md

Nodemailer

Send e-mails from Node.js – easy as cake! ✉

gitter join chat | build passing | npm package 2.6.4 | downloads 12M

Other similar packages you might be interested in

- **smtp-server** – add SMTP server interface to your application
- **smtp-connection** – connect to SMTP servers from your application
- **zone-mta** – full featured outbound MTA built using smtp-connection and smtp-server modules

Notes and information

Nodemailer supports

- **Node.js 0.10+**, no ES6 shenanigans used that would break your production app
- **Unicode** to use any characters, including full emoji support 😘
- **Windows** – you can install it with *npm* on Windows just like any other module, there are no compiled dependencies. Use it from Azure or from your Windows box hassle free.
- **HTML content** as well as **plain text** alternative
- **Attachments** (including attachment **streaming** for sending larger files)
- **Embedded images** in HTML
- Secure e-mail delivery using **SSL/STARTTLS**
- Different **transport methods**, either using built-in SMTP transports or from external plugins
- Custom **plugin support** for manipulating messages (add DKIM signatures, use markdown content instead of HTML etc.)
- Sane **XOAUTH2** login with automatic access token generation (and feedback about the updated tokens)
- Simple built-in **templating** using node-email-templates or custom renderer
- **Proxies** for SMTP connections (SOCKS, HTTP and custom connections)

> See Nodemailer homepage for complete documentation

Support Nodemailer development

Donate

If you want to support with Bitcoins, then my wallet address is `15Z8ADxhssKUiwP3jbbqJwA21744KMCfTM`

TL;DR Usage Example

This is a complete example to send an e-mail with plaintext and HTML body

```
var nodemailer = require('nodemailer');

// create reusable transporter object using the default SMTP transport
var transporter = nodemailer.createTransport('smtps://user%40gmail.com:pass@smtp.gmail.com');

// setup e-mail data with unicode symbols
var 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 🐴', // plaintext body
    html: '<b>Hello world 🐴</b>' // html body
};

// send mail with defined transport object
transporter.sendMail(mailOptions, function(error, info){
    if(error){
        return console.log(error);
    }
    console.log('Message sent: ' + info.response);
});
```

> To use Gmail you may need to configure "Allow Less Secure Apps" in your Gmail account unless you are using 2FA in which case you would have to create an Application Specific password. You also may need to unlock your account with "Allow access to your Google account" to use SMTP.

Setting up

Install with npm

```
npm install nodemailer
```

To send e-mails you need a transporter object

```
var 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

SMTP? Say what?

You might wonder why you would need to set something up while in comparison PHP's mail 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 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. Usually this works because the server is an actual web server accessible from the web and has also gathered some mail sending reputation because PHP web hosts have been around for, like, forever.

Node.js apps on the other hand might run wherever, usually on some really new VPS behind an IP address that has no sending reputation at all. Or the IP is dynamically allocated which is the fastest way to get rejected while trying to send mail. So while you might actually emulate the same behavior with Nodemailer by using either the sendmail transport or so called *direct* transport, then this does not guarantee yet any deliverability. Recipient's server might reject connection from your app because your server has dynamic IP address. Or it might reject or send your mail straight to spam mailbox because your IP address is not yet trusted.

So the reason why PHP's `mail` works and Node.js's does not is that your PHP hosting provider has put in a lot of work over several years to provide a solid mail sending infrastructure. It is not about PHP at all, it is about the infrastructure around it.

Set up SMTP

You can use 3 kinds of different approaches when using SMTP

1. *normal* usage. No specific configuration needed. For every e-mail a new SMTP connection is created and message is sent immediately. Used when the amount of sent messages is low.
2. *pooled* usage. Set *pool* option to `true` to use it. A fixed amount of pooled connections are used to send messages. Useful when you have a large number of messages that you want to send in batches.

3. *direct* usage. Set *direct* option to `true` to use it. SMTP connection is opened directly to recipients MX server, skipping any local SMTP relays. useful when you do not have a SMTP relay to use. Riskier though since messages from untrusted servers usually end up in the Spam folder.

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

Where

- **options** defines connection data
  - **options.pool** if set to `true` uses pooled connections (defaults to `false`), otherwise creates a new connection for every e-mail.
  - **options.direct** if set to `true`, bypasses MTA relay and connects directly to recipients MX. Easier to set up but has higher chances of ending up in the Spam folder
  - **options.service** can be set to the name of a well-known service so you don't have to input the `port`, `host`, and `secure` options (see Using well-known services)
  - **options.port** is the port to connect to (defaults to 25 or 465)
  - **options.host** is the hostname or IP address to connect to (defaults to `'localhost'`)
  - **options.secure** if `true` the connection will only use TLS. If `false` (the default), TLS may still be upgraded to if available via the STARTTLS command.
  - **options.ignoreTLS** if this is `true` and `secure` is false, TLS will not be used (either to connect, or as a STARTTLS connection upgrade command).
  - **options.requireTLS** if this is `true` and `secure` is false, it forces Nodemailer to use STARTTLS even if the server does not advertise support for it.
  - **options.tls** defines additional node.js TLSSocket options to be passed to the socket constructor, eg. *{rejectUnauthorized: true}*.
  - **options.auth** defines authentication data (see authentication section below)
  - **options.authMethod** defines preferred authentication method, eg. 'PLAIN'
  - **options.name** optional hostname of the client, used for identifying to the server
  - **options.localAddress** is the local interface to bind to for network connections
  - **options.connectionTimeout** how many milliseconds to wait for the connection to establish
  - **options.greetingTimeout** how many milliseconds to wait for the greeting after connection is established
  - **options.socketTimeout** how many milliseconds of inactivity to allow
  - **options.logger** optional bunyan compatible logger instance. If set to `true` then logs to console. If value is not set or is `false` then nothing is logged
  - **options.debug** if set to true, then logs SMTP traffic, otherwise logs only transaction events. This option requires **options.logger** to be set, otherwise there is nowhere to log the transaction data
  - **options.maxConnections** available only if *pool* is set to `true`. (defaults to 5) is the count of maximum simultaneous connections to make against the SMTP server
  - **options.maxMessages** available only if *pool* is set to `true`. (defaults to 100) limits the message count to be sent using a single connection. After maxMessages messages the connection is dropped and a new one is created for the following messages
  - **options.rateLimit** available only if *pool* is set to `true`. (defaults to `false`) limits the message count to be sent in a second. Once rateLimit is reached, sending is paused until the end of the second. This limit is shared between connections, so if one connection uses up the limit, then other connections are paused as well
  - **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
  - **options.disableUrlAccess** if true, then does not allow to use Urls as content

Examples

```
var smtpConfig = {
    host: 'smtp.gmail.com',
    port: 465,
    secure: true, // use SSL
    auth: {
        user: 'user@gmail.com',
        pass: 'pass'
    }
};

var poolConfig = {
    pool: true,
    host: 'smtp.gmail.com',
    port: 465,
    secure: true, // use SSL
    auth: {
        user: 'user@gmail.com',
        pass: 'pass'
    }
};

var directConfig = {
    name: 'hostname' // must be the same that can be reverse resolved by DNS for your IP
};
```

Alternatively you could use connection url. Use `smtp:` , `smtps:` or `direct:` as the protocol.

```
var smtpConfig = 'smtps://user%40gmail.com:pass@smtp.gmail.com';
var poolConfig = 'smtps://user%40gmail.com:pass@smtp.gmail.com/?pool=true';
var directConfig = 'direct:?name=hostname';
```

Proxy support

Nodemailer supports out of the box HTTP and SOCKS proxies for SMTP connections with the `proxy` configuration option. You can also use a custom connection handler with the `getSocket` method.

Proxy configuration is provided as a connection url where used protocol defines proxy protocol (eg. `'socks://hostname:port'` for a SOCKS5 proxy). You can also use authentication by passing proxy username and password into the configuration url (eg `'socks://username:password@hostname:port'`)

HTTP CONNECT tunnel

HTTP proxy must support CONNECT tunnels (also called "SSL support") to SMTP ports. To use a HTTP/S server, provide a `proxy` option to SMTP configuration with the HTTP proxy configuration URL.

```
var smtpConfig = {
    host: 'smtp.gmail.com',
    port: 465,
    ...,
    //proxy config
    // assumes a HTTP proxy running on port 3128
    proxy: 'http://localhost:3128/'
};
```

Possible protocol values for the HTTP proxy:

- `'http:'` if the proxy is running in a plaintext server
- `'https:'` if the proxy is running in a secure server

> NB! Proxy protocol (http/s) does not affect how SMTP connection is secured or not

See an example of using a HTTP proxy here.

SOCKS 4/5

To use a HTTP/S server, provide a `proxy` option to SMTP configuration with the SOCKS4/5 proxy configuration URL.

```
var smtpConfig = {
    host: 'smtp.gmail.com',
    port: 465,
    ...,
    //proxy config
    // assumes a SOCKS5 proxy running on port 1080
    proxy: 'socks5://localhost:1080/'
};
```

> **NB!** When using SOCKS4, only an ipv4 address can be used

Possible protocol values for the SOCKS proxy:

- `'socks4:'` or `'socks4a:'` for a SOCKS4 proxy
- `'socks5:'` or `'socks:'` for a SOCKS5 proxy

See an example of using a SOCKS proxy here.

Custom connection handler

If you do not want to use SOCKS or HTTP proxies then you can alternatively provide a custom proxy handling code with the `getSocket` method. In this case you should initiate a new socket yourself and pass it to Nodemailer for usage.

```
// This method is called every time Nodemailer needs a new
// connection against the SMTP server
transporter.getSocket = function(options, callback){
    getProxySocketSomehow(options.port, options.host, function(err, socket){
        if(err){
            return callback(err);
        }
        callback(null, {
            connection: socket
        });
    });
};
```

Normally proxies provide plaintext sockets, so if the connection is supposed to use TLS then Nodemailer upgrades the socket from plaintext to TLS itself. If the socket is already upgraded then you can pass additional option `secured: true` to prevent Nodemailer from upgrading the already upgraded socket.

```
callback(null, {
    connection: socket,
    secured: true
});
```

See complete example using a custom socket connector here.

Events

Event:'idle'

Applies to pooled SMTP connections. Emitted by the transport 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 like RabbitMQ.

```
var messages = [...'list of messages'];
transporter.on('idle', function(){
    // send next messages from the pending queue
    while(transporter.isIdle() && messages.length){
        transporter.send(messages.shift());
    }
});
```

Authentication

If authentication data is not present, the connection is considered authenticated from the start. Set authentication data with `options.auth`

- **auth** is the authentication object
  - **auth.user** is the username
  - **auth.pass** is the password for the user
  - **auth.xoauth2** is the OAuth2 access token (preferred if both `pass` and `xoauth2` values are set) or an XOAuth2 token generator object.

Using OAuth2

If a XOAuth2 token generator is used as the value for `auth.xoauth2` then you do not need to set the value for `user` or `pass`. XOAuth2 generator generates required `accessToken` itself if it is missing or expired. In this case if the authentication fails, a new token is requested and the authentication is retried once. If it still fails, an error is returned.

> **NB!** The correct OAuth2 scope for Gmail is `https://mail.google.com/`

Install xoauth2 module to use XOauth2 token generators (not included by default)

```
npm install xoauth2 --save
```

Example

```
var nodemailer = require('nodemailer');
var xoauth2 = require('xoauth2');

// listen for token updates (if refreshToken is set)
// you probably want to store these to a db
generator.on('token', function(token){
    console.log('New token for %s: %s', token.user, token.accessToken);
});

// login
var transporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
        xoauth2: xoauth2.createXOAuth2Generator({
            user: '{username}',
            clientId: '{Client ID}',
            clientSecret: '{Client Secret}',
            refreshToken: '{refresh-token}',
            accessToken: '{cached access token}'
        })
    }
});
```

Using *well-known* services

If you do not want to specify the hostname, port and security settings for a well known service, you can use it by its name (case insensitive)

```
smtpTransport({
    service: 'gmail',
    auth: ..
});
```

See the list of all supported services here.

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');
   }
});
```

Send using a transport plugin

In addition to SMTP you can use other kind of transports as well with Nodemailer. See *Available Transports* below for known transports.

The following example uses nodemailer-ses-transport (Amazon SES).

```
var nodemailer = require('nodemailer');
var ses = require('nodemailer-ses-transport');
var transporter = nodemailer.createTransport(ses({
    accessKeyId: 'AWSACCESSKEY',
    secretAccessKey: 'AWS/Secret/key'
}));
```

If the transport plugin follows common conventions, then you can also load it dynamically with the `transport` option. This way you would not have to load the transport plugin in your code (you do need to install the transport plugin though before you can use it), you only need to modify the configuration data accordingly.

```
var nodemailer = require('nodemailer');
var transporter = nodemailer.createTransport({
    transport: 'ses', // loads nodemailer-ses-transport
    accessKeyId: 'AWSACCESSKEY',
    secretAccessKey: 'AWS/Secret/key'
});
```

Available Transports

- **nodemailer-mailgun-transport** for sending messages through Mailgun's Web API
- **nodemailer-mandrill-transport** for sending messages through Mandrill's Web API
- **nodemailer-pickup-transport** for storing messages to pickup folders
- **nodemailer-sailthru-transport** for sending messages through Sailthru's Web API
- **nodemailer-sendgrid-transport** for sending messages through SendGrid's Web API
- **nodemailer-sendmail-transport** for piping messages to the *sendmail* command
- **nodemailer-ses-transport** for sending messages to AWS SES
- **nodemailer-sparkpost-transport** for sending messages through SparkPost's Web API

- **nodemailer-stub-transport** is just for returning messages, most probably for testing purposes
- **nodemailer-wellknown** for sending messages through one of those many supported services
- **nodemailer-postmark-transport** for sending messages through Postmark's Web API
- *add yours* (see transport api documentation here)

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 e-mail message fields below)
- **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
    - **info.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.

E-mail message fields

The following are the possible fields of an e-mail message:

Commmon fields:

- **from** - The e-mail address of the sender. All e-mail addresses can be plain `'sender@server.com'` or formatted `'"Sender Name" <sender@server.com>'`, see Address Formatting for details
- **to** - Comma separated list or an array of recipients e-mail addresses that will appear on the *To:* field
- **cc** - Comma separated list or an array of recipients e-mail addresses that will appear on the *Cc:* field
- **bcc** - Comma separated list or an array of recipients e-mail addresses that will appear on the *Bcc:* field
- **subject** - The subject of the e-mail
- **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 below for details)

Advanced fields:

- **sender** - An e-mail address that will appear on the *Sender:* field (always prefer `from` if you're not sure which one to use)
- **replyTo** - An e-mail 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)
- **watchHtml** - Apple Watch specific HTML version of the message. Same usage as with `text` or `html`
- **icalEvent** – iCalendar event to use as an alternative. Same usage as with `text` or `html`. Additionally you could set `method` property (defaults to `'PUBLISH'`). See an example here
- **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"}]*)
- **alternatives** - An array of alternative text contents (in addition to text and html parts) (see below for details)
- **envelope** - optional SMTP envelope, if auto generated envelope is not suitable (see below for details)
- **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
- **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. If this value is set then you should also set the envelope object (if required) as the provided raw message is not parsed. The value could be a string, a buffer, a stream or an attachment-like object.
- **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*)
- **list** - helper for setting List-* headers

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

> **NB!** When using readable streams as any kind of content and sending fails then Nodemailer does not abort the already opened but not yet finished stream automatically, you need to do this yourself

```
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

Attachment object consists of the following properties:

- **filename** - filename to be reported as the name of the attached file, use of unicode is allowed. If you do not want to use a filename, set this value as `false`, otherwise a filename is generated automatically
- **content** - String, Buffer or a Stream contents for the attachment
- **path** - path to a file or an URL (data uris are allowed as well) if you want to stream the file instead of including it (better for larger attachments)
- **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 e-mail 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

```
var mailOptions = {
    ...
    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='
        }
    ]
}
```

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 e-mail client to select and show the best fitting alternative to the reader. Usually this field is used for calendar events and such.

Alternative objects use the same options as attachment objects. 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:

```
var mailOptions = {
    ...
    html: '<b>Hello world!</b>',
    alternatives: [
        {
            contentType: 'text/x-web-markdown',
            content: '**Hello world!**'
        }
    ]
}
```

Alternatives can be added as many as you want.

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.

```
var mail = {
    ...,
    headers: {
        'x-my-key': 'header value',
        'x-another-key': 'another value'
    }
}

// X-My-Key: header value
// X-Another-Key: another value
```

Multiple rows

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

```
var mail = {
    ...,
    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
```

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:

```
var mail = {
    ...,
    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 non-ascii characters 👮
```

Address Formatting

All the e-mail addresses can be plain e-mail 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 in the name part, make sure you enclose the name in double quotes: `"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 comma separated list of e-mails or an array of e-mails or an array of comma separated list of e-mails 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>'
```

SMTP envelope

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

`envelope` is an object with the following params: `from` , `to` , `cc` and `bcc` just like with regular mail options. You can also use the regular address format, unicode domains etc.

```
mailOptions = {
    ...,
    from: 'mailer@kreata.ee', // listed in rfc822 message header
    to: 'daemon@kreata.ee', // listed in rfc822 message header
    envelope: {
        from: '"Daemon" <deamon@kreata.ee>', // used as MAIL FROM: address for SMTP
        to: 'mailer@kreata.ee, "Mailer" <mailer2@kreata.ee>' // used as RCPT TO: address for SMTP
    }
}
```

> Not all transports can use the `envelope` object, for example SES ignores it and only uses the data from the From:, To: etc. headers.

Using Embedded Images

A new alternative to cid embedded images is available! See nodemailer-base64-to-s3 for more information.

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!

```
var mailOptions = {
    ...
    html: 'Embedded image: <img src="cid:unique@kreata.ee"/>',
    attachments: [{
        filename: 'image.png',
        path: '/path/to/file',
        cid: 'unique@kreata.ee' //same cid value as in the html img src
    }]
}
```

Using templates

Nodemailer allows to use simple built-in templating or alternatively external renderers for common message types.

```
var transporter = nodemailer.createTransport(...);
var send = transporter.templateSender(templates, [defaults]);

// send a message based on provided templates
send(mailData, context, callback);
// or
send(mailData, context).then(...).catch(...);
```

Where

- **templates** is an object with template strings for built-in renderer or an EmailTemplate object for more complex rendering

```
// built-in renderer
var send = transporter.templateSender({
    subject: 'This template is used for the "subject" field',
    text: 'This template is used for the "text" field',
    html: 'This template is used for the "html" field'
});
// external renderer
var EmailTemplate = require('email-templates').EmailTemplate;
var send = transporter.templateSender(new EmailTemplate('template/directory'));
```

- **defaults** is an optional object of message data fields that are set for every message sent using this sender
- **mailData** includes message fields for current message
- **context** is an object with template replacements, where `key` replaces `{{key}}` when using the built-in renderer

```
var templates = {
    text: 'Hello {{username}}!'
};
var context = {
    username: 'User Name'
};
// results in "Hello, User Name!" as the text body
// of the message when using built-in renderer
```

- **callback** is the `transporter.sendMail` callback (if not set then the function returns a Promise)

> **NB!** If using built-in renderer then template variables are HTML escaped for the `html` field but kept as is for other fields

Example 1. Built-in renderer

```
var transporter = nodemailer.createTransport('smtps://user%40gmail.com:pass@smtp.gmail.com');

// create template based sender function
var sendPwdReset = transporter.templateSender({
    subject: 'Password reset for {{username}}!',
    text: 'Hello, {{username}}, Please go here to reset your password: {{ reset }}',
    html: '<b>Hello, <strong>{{username}}</strong>, Please <a href="{{ reset }}">go here to reset your password</a>:
}, {
    from: 'sender@example.com',
});

// use template based sender to send a message
sendPwdReset({
    to: 'receiver@example.com'
}, {
    username: 'Node Mailer',
    reset: 'https://www.example.com/reset?token=<unique-single-use-token>'
}, function(err, info){
    if(err){
        console.log('Error');
    }else{
        console.log('Password reset sent');
    }
});
```

Example 2. External renderer

```
var EmailTemplate = require('email-templates').EmailTemplate;
var transporter = nodemailer.createTransport('smtps://user%40gmail.com:pass@smtp.gmail.com');

// create template based sender function
// assumes text.{ext} and html.{ext} in template/directory
var sendPwdReminder = transporter.templateSender(new EmailTemplate('template/directory'), {
    from: 'sender@example.com',
});

// use template based sender to send a message
sendPwdReminder({
    to: 'receiver@example.com',
    // EmailTemplate renders html and text but no subject so we need to
    // set it manually either here or in the defaults section of templateSender()
    subject: 'Password reminder'
}, {
    username: 'Node Mailer',
    password: '!"\'<>&some-thing'
}, function(err, info){
    if(err){
       console.log('Error');
    }else{
        console.log('Password reminder sent');
    }
});
```

Custom renderer

In addition to the built-in and node-email-templates based renderers you can also bring your own.

```
var sendPwdReminder = transporter.templateSender({
    render: function(context, callback){
        callback(null, {
            html: 'rendered html content',
            text: 'rendered text content'
        });
    }
});
```

Example. Using swig-email-templates

```
var EmailTemplates = require('swig-email-templates');
var transporter = nodemailer.createTransport('smtps://user%40gmail.com:pass@smtp.gmail.com');

// create template renderer
var templates = new EmailTemplates();

// provide custom rendering function
var sendPwdReminder = transporter.templateSender({
    render: function(context, callback){
        templates.render('pwreminder.html', context, function (err, html, text) {
            if(err){
                return callback(err);
            }
            callback(null, {
                html: html,
                text: text
            });
        });
    }
});
...
```

List-* headers

Nodemailer includes a helper for setting more complex List-* headers with ease. Use message option `list` to provide all list headers. You do not need to add protocol prefix for the urls, or enclose the url between < and >, this is handled automatically.

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.

```
var mailOptions = {
    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'
                }
            ]
```

```
        ]
    }
};
```

Available Plugins

In addition to built-in e-mail fields you can extend these by using plugins.

- **nodemailer-base64-to-s3** to convert your Base64-Encoded Data URI's in `<img>` tags to Amazon S3/CloudFront URL's (an alternative to cid embedded images)
- **nodemailer-markdown** to use markdown for the content
- **nodemailer-dkim** to sign messages with DKIM
- **nodemailer-html-to-text** to auto generate plaintext content from html
- **nodemailer-express-handlebars** to auto generate html emails from handlebars/mustache templates
- **nodemailer-plugin-inline-base64** to convert base64 images to attachments
- **nodemailer-hashcash** to generate hashcash headers
- **nodemailer-trap-plugin** to intercept emails in non production environments
- *add yours* (see plugin api documentation here)

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' 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 (support for 'less secure' apps is enabled) but other is blocked (support for 'less secure' apps is disabled). When using this method make sure to enable the required functionality by completing the "captcha enable". Without this, less secure connections won't work.

To prevent having login issues you should either use XOAUTH2 (see details here) or use another provider and preferably a dedicated one like Mailgun or SendGrid or any other. 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 (originally published as a blog post).

1. **Use a dedicated SMTP provider** like SendGrid or Mailgun or any other. 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 immediatelly or get labelled as spam. Basically you get what you pay for and if you pay zero then your deliverability is near zero as well. E-mail might seem free but it is only free to a certain amount and that amount certainly does not include 10 million e-mails in a short period of time.
2. **Use a dedicated queue manager,** for example RabbitMQ for queueing the e-mails. 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). See rabbit-queue for an example of using RabbitMQ queues with Nodemailer connection pool.

3. **Use nodemailer-smtp-pool transport.** You do not want to have the overhead of creating a new connection and doing the SMTP handshake dance for every single e-mail. Pooled connections make it possible to bring this overhead to a minimum.
4. **Set `maxMessages` option to `Infinity`** for the nodemailer-smtp-pool transport. Dedicated SMTP providers happily accept all your e-mails 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 e-mails itself, you can't beat this with SMTP.

Implementing plugins and transports

There are 3 stages a plugin can hook to

1. **'compile'** is the step where e-mail data is set but nothing has been done with it yet. At this step you can modify mail options, for example modify `html` content, add new headers etc. Example: nodemailer-markdown that allows you to use `markdown` source instead of `text` and `html`.
2. **'stream'** is the step where message tree has been compiled and is ready to be streamed. At this step you can modify the generated MIME tree or add a transform stream that the generated raw e-mail will be piped through before passed to the transport object. Example: nodemailer-dkim that adds DKIM signature to the generated message.
3. **Transport** step where the raw e-mail is streamed to destination. Example: nodemailer-smtp-transport that streams the message to a SMTP server.

Including plugins

'compile' and 'stream' plugins can be attached with `use(plugin)` method

```
transporter.use(step, pluginFunc)
```

Where

- **transporter** is a transport object created with `createTransport`
- **step** is a string, either 'compile' or 'stream' that defines when the plugin should be hooked
- **pluginFunc** is a function that takes two arguments: the mail object and a callback function

Plugin API

All plugins (including transports) get two arguments, the mail object and a callback function.

Mail object that is passed to the plugin function as the first argument is an object with the following properties:

- **data** is the mail data object that is passed to the `sendMail` method
- **message** is the BuildMail object of the message. This is available for the 'stream' step and for the transport but not for 'compile'.

- **resolveContent** is a helper function for converting Nodemailer compatible stream objects into Strings or Buffers

resolveContent()

If your plugin needs to get the full value of a param, for example the String value for the `html` content, you can use `resolveContent()` to convert Nodemailer compatible content objects to Strings or Buffers.

```
data.resolveContent(obj, key, callback)
```

Where

- **obj** is an object that has a property you want to convert to a String or a Buffer
- **key** is the name of the property you want to convert
- **callback** is the callback function with (err, value) where `value` is either a String or Buffer, depending on the input

Example

```
function plugin(mail, callback){
    // if mail.data.html is a file or an url, it is returned as a Buffer
    mail.resolveContent(mail.data, 'html', function(err, html){
        if(err){
            return callback(err);
        }
        console.log('HTML contents: %s', html.toString());
        callback();
    });
};
```

'compile'

Compile step plugins get only the `mail.data` object but not `mail.message` in the `mail` argument of the plugin function. If you need to access the `mail.message` as well use 'stream' step instead.

This is really straightforward, your plugin can modify the `mail.data` object at will and once everything is finished run the callback function. If the callback gets an error object as an argument, then the process is terminated and the error is returned to the `sendMail` callback.

Example

The following plugin checks if `text` value is set and if not converts `html` value to `text` by removing all html tags.

```
transporter.use('compile', function(mail, callback){
    if(!mail.text && mail.html){
        mail.text = mail.html.replace(/<[^>]*>/g, ' ');
    }
    callback();
});
```

See plugin-compile.js for a working example.

'stream'

Streaming step is invoked once the message structure is built and ready to be streamed to the transport. Plugin function still gets `mail.data` but it is included just for the reference, modifying it should not change anything (unless the transport requires something from the `mail.data`, for example `mail.data.envelope`).

You can modify the `mail.message` object as you like, the message is not yet streaming anything (message starts streaming when the transport calls `mail.message.createReadStream()`).

In most cases you might be interested in the message.transform() method for applying transform streams to the raw message.

Example

The following plugin replaces all tabs with spaces in the raw message.

```
var transformer = new (require('stream').Transform)();
transformer._transform = function(chunk, encoding, done) {
    // replace all tabs with spaces in the stream chunk
    for(var i = 0; i < chunk.length; i++){
        if(chunk[i] === 0x09){
            chunk[i] = 0x20;
        }
    }
    this.push(chunk);
    done();
};

transporter.use('stream', function(mail, callback){
    // apply output transformer to the raw message stream
    mail.message.transform(transformer);
    callback();
});
```

See plugin-stream.js for a working example.

Additionally you might be interested in the message.getAddresses() method that returns the contents for all address fields as structured objects.

Example

The following plugin prints address information to console.

```
transporter.use('stream', function(mail, callback){
    var addresses = mail.message.getAddresses();
    console.log('From: %s', JSON.stringify(addresses.from));
    console.log('To: %s', JSON.stringify(addresses.to));
    console.log('Cc: %s', JSON.stringify(addresses.cc));
    console.log('Bcc: %s', JSON.stringify(addresses.bcc));
    callback();
});
```

Transports

Transports are objects that have a method `send` and properies `name` and `version`. Additionally, if the transport object is an Event Emitter, 'log' events are piped through Nodemailer. A transport object is passed to the `nodemailer.createTransport(transport)` method to create the transporter object.

`transport.name`

This is the name of the transport object. For example 'SMTP' or 'SES' etc.

```
transport.name = require('package.json').name;
```

`transport.version`

This should be the transport module version. For example '0.1.0'.

```
transport.version = require('package.json').version;
```

`transport.send(mail, callback)`

This is the method that actually sends out e-mails. The method is basically the same as 'stream' plugin functions. It gets two arguments: `mail` and a callback. To start streaming the message, create the stream with `mail.message.createReadStream()`

Callback function should return an `info` object as the second arugment. This info object should contain `messageId` value with the Message-Id header (without the surrounding < > brackets)

The following example pipes the raw stream to the console.

```
transport.send = function(mail, callback){
    var input = mail.message.createReadStream();
    var messageId = (mail.message.getHeader('message-id') || '').replace(/[<>\s]/g, '');
    input.pipe(process.stdout);
    input.on('end', function() {
        callback(null, {
            messageId: messageId
        });
    });
};
```

`transport.close(args*)`

If your transport needs to be closed explicitly, you can implement a `close` method.

This is purely optional feature and only makes sense in special contexts (eg. closing a SMTP pool).

`transport.isIdle()`

If your transport is able to notify about idling state by issuing `'idle'` events then this method should return if the transport is still idling or not.

Wrapping up

Once you have a transport object, you can create a mail transporter out of it.

```
var nodemailer = require('nodemailer');
var transport = require('some-transport-method');
var transporter = nodemailer.createTransport(transport);
transporter.sendMail({mail data});
```

See minimal-transport.js for a working example.

License

Nodemailer is licensed under MIT license. Basically you can do whatever you want to with it

The Nodemailer logo was designed by Sven Kristjansen.