1151 lines (1021 sloc) | 39.3 KB

```
'use strict';

const mailsplit = require('mailsplit');
const libmime = require('libmime');
const addressparser = require('nodemailer/lib/addressparser');
const Transform = require('stream').Transform;
const Splitter = mailsplit.Splitter;
const punycode = require('punycode');
const FlowedDecoder = require('mailsplit/lib/flowed-decoder');
const StreamHash = require('./stream-hash');
const iconv = require('iconv-lite');
const { htmlToText } = require('html-to-text');
const he = require('he');
const linkify = require('linkify-it')();
const tlds = require('tlds');
const encodingJapanese = require('encoding-japanese');

linkify
    .tlds(tlds) // Reload with full tlds list
    .tlds('onion', true) // Add unofficial `.onion` domain
    .add('git:', 'http:') // Add `git:` ptotocol as "alias"
    .add('ftp:', null) // Disable `ftp:` ptotocol
    .set({ fuzzyIP: true, fuzzyLink: true, fuzzyEmail: true });

// twitter linkifier from
// https://github.com/markdown-it/linkify-it#example-2-add-twitter-mentions-handler
linkify.add('@', {
    validate(text, pos, self) {
        let tail = text.slice(pos);
```

```
        if (!self.re.twitter) {
            self.re.twitter = new RegExp('^([a-zA-Z0-9_]){1,15}(?!_)(?=$|' + self.re.src_Z
        }
        if (self.re.twitter.test(tail)) {
            // Linkifier allows punctuation chars before prefix,
            // but we additionally disable `@` ("@@mention" is invalid)
            if (pos >= 2 && tail[pos - 2] === '@') {
                return false;
            }
            return tail.match(self.re.twitter)[0].length;
        }
        return 0;
    },
    normalize(match) {
        match.url = 'https://twitter.com/' + match.url.replace(/^@/, '');
    }
});

class IconvDecoder extends Transform {
    constructor(Iconv, charset) {
        super();

        // Iconv throws error on ks_c_5601-1987 when it is mapped to EUC-KR
        // https://github.com/bnoordhuis/node-iconv/issues/169
        if (charset.toLowerCase() === 'ks_c_5601-1987') {
            charset = 'CP949';
        }
        this.stream = new Iconv(charset, 'UTF-8//TRANSLIT//IGNORE');

        this.inputEnded = false;
        this.endCb = false;

        this.stream.on('error', err => this.emit('error', err));
        this.stream.on('data', chunk => this.push(chunk));
        this.stream.on('end', () => {
            this.inputEnded = true;
            if (typeof this.endCb === 'function') {
                this.endCb();
            }
        });
    }

    _transform(chunk, encoding, done) {
        this.stream.write(chunk);
        done();
    }

    _flush(done) {
```

```
        this.endCb = done;
        this.stream.end();
    }
}

class JPDecoder extends Transform {
    constructor(charset) {
        super();

        this.charset = charset;
        this.chunks = [];
        this.chunklen = 0;
    }

    _transform(chunk, encoding, done) {
        if (typeof chunk === 'string') {
            chunk = Buffer.from(chunk, encoding);
        }

        this.chunks.push(chunk);
        this.chunklen += chunk.length;
        done();
    }

    _flush(done) {
        let input = Buffer.concat(this.chunks, this.chunklen);
        try {
            let output = encodingJapanese.convert(input, {
                to: 'UNICODE', // to_encoding
                from: this.charset, // from_encoding
                type: 'string'
            });
            if (typeof output === 'string') {
                output = Buffer.from(output);
            }
            this.push(output);
        } catch (err) {
            // keep as is on errors
            this.push(input);
        }

        done();
    }
}

class MailParser extends Transform {
    constructor(config) {
        super({
            readableObjectMode: true,
```

```
            writableObjectMode: false
        });

        this.options = config || {};
        this.splitter = new Splitter(config);
        this.finished = false;
        this.waitingEnd = false;

        this.headers = false;
        this.headerLines = false;

        this.endReceived = false;
        this.reading = false;
        this.errored = false;

        this.tree = false;
        this.curnode = false;
        this.waitUntilAttachmentEnd = false;
        this.attachmentCallback = false;

        this.hasHtml = false;
        this.hasText = false;

        this.text = false;
        this.html = false;
        this.textAsHtml = false;

        this.attachmentList = [];

        this.boundaries = [];

        this.textTypes = ['text/plain', 'text/html'].concat(!this.options.keepDeliveryStat

        this.decoder = this.getDecoder();

        this.splitter.on('readable', () => {
            if (this.reading) {
                return false;
            }
            this.readData();
        });

        this.splitter.on('end', () => {
            this.endReceived = true;
            if (!this.reading) {
                this.endStream();
            }
        });

```

```
        this.splitter.on('error', err => {
            this.errored = true;
            if (typeof this.waitingEnd === 'function') {
                return this.waitingEnd(err);
            }
            this.emit('error', err);
        });

        this.libmime = new libmime.Libmime({ Iconv: this.options.Iconv });
    }

    getDecoder() {
        if (this.options.Iconv) {
            const Iconv = this.options.Iconv;
            // create wrapper
            return {
                decodeStream(charset) {
                    return new IconvDecoder(Iconv, charset);
                }
            };
        } else {
            return {
                decodeStream(charset) {
                    charset = (charset || 'ascii').toString().trim().toLowerCase();
                    if (/^jis|^iso-?2022-?jp|^EUCJP/i.test(charset)) {
                        // special case not supported by iconv-lite
                        return new JPDecoder(charset);
                    }

                    return iconv.decodeStream(charset);
                }
            };
        }
    }

    readData() {
        if (this.errored) {
            return false;
        }
        this.reading = true;
        let data = this.splitter.read();
        if (data === null) {
            this.reading = false;
            if (this.endReceived) {
                this.endStream();
            }
            return;
        }

```

```
        this.processChunk(data, err => {
            if (err) {
                if (typeof this.waitingEnd === 'function') {
                    return this.waitingEnd(err);
                }
                return this.emit('error', err);
            }
            setImmediate(() => this.readData());
        });
    }

    endStream() {
        this.finished = true;

        if (this.curnode && this.curnode.decoder) {
            this.curnode.decoder.end();
        }
        if (typeof this.waitingEnd === 'function') {
            this.waitingEnd();
        }
    }

    _transform(chunk, encoding, done) {
        if (!chunk || !chunk.length) {
            return done();
        }

        if (this.splitter.write(chunk) === false) {
            return this.splitter.once('drain', () => {
                done();
            });
        } else {
            return done();
        }
    }

    _flush(done) {
        setImmediate(() => this.splitter.end());
        if (this.finished) {
            return this.cleanup(done);
        }
        this.waitingEnd = () => {
            this.cleanup(() => {
                done();
            });
        };
    }

    cleanup(done) {
```

```
        let finish = () => {
            let t = this.getTextContent();
            this.push(t);
            done();
        };

        if (this.curnode && this.curnode.decoder && this.curnode.decoder.readable && !this
            (this.curnode.contentStream || this.curnode.decoder).once('end', () => {
                finish();
            });
            this.curnode.decoder.end();
        } else {
            setImmediate(() => {
                finish();
            });
        }
    }

    processHeaders(lines) {
        let headers = new Map();
        (lines || []).forEach(line => {
            let key = line.key;
            let value = ((this.libmime.decodeHeader(line.line) || {}).value || '').toStrin
            value = Buffer.from(value, 'binary').toString();
            switch (key) {
                case 'content-type':
                case 'content-disposition':
                case 'dkim-signature':
                    value = this.libmime.parseHeaderValue(value);
                    if (value.value) {
                        value.value = this.libmime.decodeWords(value.value);
                    }
                    Object.keys((value && value.params) || {}).forEach(key => {
                        try {
                            value.params[key] = this.libmime.decodeWords(value.params[key]
                        } catch (E) {
                            // ignore, keep as is
                        }
                    });
                    break;
                case 'date': {
                    let dateValue = new Date(value);
                    if (isNaN(dateValue)) {
                        // date parsing failed :S
                        dateValue = new Date();
                    }
                    value = dateValue;
                    break;
                }
```

```
                case 'subject':
                    try {
                        value = this.libmime.decodeWords(value);
                    } catch (E) {
                        // ignore, keep as is
                    }
                    break;
                case 'references':
                    try {
                        value = this.libmime.decodeWords(value);
                    } catch (E) {
                        // ignore
                    }
                    value = value.split(/\s+/).map(this.ensureMessageIDFormat);
                    break;
                case 'message-id':
                case 'in-reply-to':
                    try {
                        value = this.libmime.decodeWords(value);
                    } catch (E) {
                        // ignore
                    }
                    value = this.ensureMessageIDFormat(value);
                    break;
                case 'priority':
                case 'x-priority':
                case 'x-msmail-priority':
                case 'importance':
                    key = 'priority';
                    value = this.parsePriority(value);
                    break;
                case 'from':
                case 'to':
                case 'cc':
                case 'bcc':
                case 'sender':
                case 'reply-to':
                case 'delivered-to':
                case 'return-path':
                    value = addressparser(value);
                    this.decodeAddresses(value);
                    value = {
                        value,
                        html: this.getAddressesHTML(value),
                        text: this.getAddressesText(value)
                    };
                    break;
            }

```

```
373             // handle list-* keys
374             if (key.substr(0, 5) === 'list-') {
375                 value = this.parseListHeader(key.substr(5), value);
376                 key = 'list';
377             }
378
379             if (value) {
380                 if (!headers.has(key)) {
381                     headers.set(key, [].concat(value || []));
382                 } else if (Array.isArray(value)) {
383                     headers.set(key, headers.get(key).concat(value));
384                 } else {
385                     headers.get(key).push(value);
386                 }
387             }
388         });
389
390         // keep only the first value
391         let singleKeys = [
392             'message-id',
393             'content-id',
394             'from',
395             'sender',
396             'in-reply-to',
397             'reply-to',
398             'subject',
399             'date',
400             'content-disposition',
401             'content-type',
402             'content-transfer-encoding',
403             'priority',
404             'mime-version',
405             'content-description',
406             'precedence',
407             'errors-to'
408         ];
409
410         headers.forEach((value, key) => {
411             if (Array.isArray(value)) {
412                 if (singleKeys.includes(key) && value.length) {
413                     headers.set(key, value[value.length - 1]);
414                 } else if (value.length === 1) {
415                     headers.set(key, value[0]);
416                 }
417             }
418
419             if (key === 'list') {
420                 // normalize List-* headers
421                 let listValue = {};
```

```
                [].concat(value || []).forEach(val => {
                    Object.keys(val || {}).forEach(listKey => {
                        listValue[listKey] = val[listKey];
                    });
                });
                headers.set(key, listValue);
            }
        });

        return headers;
    }

    parseListHeader(key, value) {
        let addresses = addressparser(value);
        let response = {};
        let data = addresses
            .map(address => {
                if (/^https?:/i.test(address.name)) {
                    response.url = address.name;
                } else if (address.name) {
                    response.name = address.name;
                }
                if (/^mailto:/.test(address.address)) {
                    response.mail = address.address.substr(7);
                } else if (address.address && address.address.indexOf('@') < 0) {
                    response.id = address.address;
                } else if (address.address) {
                    response.mail = address.address;
                }
                if (Object.keys(response).length) {
                    return response;
                }
                return false;
            })
            .filter(address => address);
        if (data.length) {
            return {
                [key]: response
            };
        }
        return false;
    }

    parsePriority(value) {
        value = value.toLowerCase().trim();
        if (!isNaN(parseInt(value, 10))) {
            // support "X-Priority: 1 (Highest)"
            value = parseInt(value, 10) || 0;
            if (value === 3) {
```

```
                return 'normal';
            } else if (value > 3) {
                return 'low';
            } else {
                return 'high';
            }
        } else {
            switch (value) {
                case 'non-urgent':
                case 'low':
                    return 'low';
                case 'urgent':
                case 'high':
                    return 'high';
            }
        }
        return 'normal';
    }

    ensureMessageIDFormat(value) {
        if (!value.length) {
            return false;
        }

        if (value.charAt(0) !== '<') {
            value = '<' + value;
        }

        if (value.charAt(value.length - 1) !== '>') {
            value += '>';
        }

        return value;
    }

    decodeAddresses(addresses) {
        for (let i = 0; i < addresses.length; i++) {
            let address = addresses[i];
            address.name = (address.name || '').toString().trim();

            if (!address.address && /^(=\?([^?]+)\?[Bb]\?[^?]*\?=)(\s*=\?([^?]+)\?[Bb]\?[^
                let parsed = addressparser(this.libmime.decodeWords(address.name));
                if (parsed.length) {
                    parsed.forEach(entry => addresses.push(entry));
                }

                // remove current element
                addresses.splice(i, 1);
                i--;
```

```
                continue;
            }

            if (address.name) {
                try {
                    address.name = this.libmime.decodeWords(address.name);
                } catch (E) {
                    //ignore, keep as is
                }
            }
            if (/@xn--/.test(address.address)) {
                try {
                    address.address =
                        address.address.substr(0, address.address.lastIndexOf('@') + 1) +
                        punycode.toUnicode(address.address.substr(address.address.lastInde
                } catch (E) {
                    // Not a valid punycode string; keep as is
                }
            }
            if (address.group) {
                this.decodeAddresses(address.group);
            }
        }
    }

    createNode(node) {
        let contentType = node.contentType;
        let disposition = node.disposition;
        let encoding = node.encoding;
        let charset = node.charset;

        if (!contentType && node.root) {
            contentType = 'text/plain';
        }

        let newNode = {
            node,
            headerLines: node.headers.lines,
            headers: this.processHeaders(node.headers.getList()),
            contentType,
            children: []
        };

        if (!/^multipart\//i.test(contentType)) {
            if (disposition && !['attachment', 'inline'].includes(disposition)) {
                disposition = 'attachment';
            }

            if (!disposition && !this.textTypes.includes(contentType)) {
```

```
569                 newNode.disposition = 'attachment';
570             } else {
571                 newNode.disposition = disposition || 'inline';
572             }
573
574             newNode.isAttachment = !this.textTypes.includes(contentType) || newNode.dispos
575
576             newNode.encoding = ['quoted-printable', 'base64'].includes(encoding) ? encodin
577
578             if (charset) {
579                 newNode.charset = charset;
580             }
581
582             let decoder = node.getDecoder();
583             decoder.on('end', () => {
584                 this.decoderEnded = true;
585             });
586             newNode.decoder = decoder;
587         }
588
589         if (node.root) {
590             this.headers = newNode.headers;
591             this.headerLines = newNode.headerLines;
592         }
593
594         // find location in tree
595
596         if (!this.tree) {
597             newNode.root = true;
598             this.curnode = this.tree = newNode;
599             return newNode;
600         }
601
602         // immediate child of root node
603         if (!this.curnode.parent) {
604             newNode.parent = this.curnode;
605             this.curnode.children.push(newNode);
606             this.curnode = newNode;
607             return newNode;
608         }
609
610         // siblings
611         if (this.curnode.parent.node === node.parentNode) {
612             newNode.parent = this.curnode.parent;
613             this.curnode.parent.children.push(newNode);
614             this.curnode = newNode;
615             return newNode;
616         }
617
```

```
        // first child
        if (this.curnode.node === node.parentNode) {
            newNode.parent = this.curnode;
            this.curnode.children.push(newNode);
            this.curnode = newNode;
            return newNode;
        }

        // move up
        let parentNode = this.curnode;
        while ((parentNode = parentNode.parent)) {
            if (parentNode.node === node.parentNode) {
                newNode.parent = parentNode;
                parentNode.children.push(newNode);
                this.curnode = newNode;
                return newNode;
            }
        }

        // should never happen, can't detect parent
        this.curnode = newNode;
        return newNode;
    }

    getTextContent() {
        let text = [];
        let html = [];
        let processNode = (alternative, level, node) => {
            if (node.showMeta) {
                let meta = ['From', 'Subject', 'Date', 'To', 'Cc', 'Bcc']
                    .map(fkey => {
                        let key = fkey.toLowerCase();
                        if (!node.headers.has(key)) {
                            return false;
                        }
                        let value = node.headers.get(key);
                        if (!value) {
                            return false;
                        }
                        return {
                            key: fkey,
                            value: Array.isArray(value) ? value[value.length - 1] : value
                        };
                    })
                    .filter(entry => entry);
                if (this.hasHtml) {
                    html.push(
                        '<table class="mp_head">' +
                            meta
```

```
667                                 .map(entry => {
668                                     let value = entry.value;
669                                     switch (entry.key) {
670                                         case 'From':
671                                         case 'To':
672                                         case 'Cc':
673                                         case 'Bcc':
674                                             value = value.html;
675                                             break;
676                                         case 'Date':
677                                             value = this.options.formatDateString ? this.o
678                                             break;
679                                         case 'Subject':
680                                             value = '<strong>' + he.encode(value) + '</str
681                                             break;
682                                         default:
683                                             value = he.encode(value);
684                                     }
685 
686                                     return '<tr><td class="mp_head_key">' + he.encode(entr
687                                 })
688                                 .join('\n') +
689                             '<table>'
690                         );
691                     }
692                     if (this.hasText) {
693                         text.push(
694                             '\n' +
695                                 meta
696                                     .map(entry => {
697                                         let value = entry.value;
698                                         switch (entry.key) {
699                                             case 'From':
700                                             case 'To':
701                                             case 'Cc':
702                                             case 'Bcc':
703                                                 value = value.text;
704                                                 break;
705                                             case 'Date':
706                                                 value = this.options.formatDateString ? this.o
707                                                 break;
708                                         }
709                                         return entry.key + ': ' + value;
710                                     })
711                                     .join('\n') +
712                             '\n'
713                         );
714                     }
715                 }
```

```
            if (node.textContent) {
                if (node.contentType === 'text/plain') {
                    text.push(node.textContent);
                    if (!alternative && this.hasHtml) {
                        html.push(this.textToHtml(node.textContent));
                    }
                } else if (node.contentType === 'message/delivery-status' && !this.options
                    text.push(node.textContent);
                    if (!alternative && this.hasHtml) {
                        html.push(this.textToHtml(node.textContent));
                    }
                } else if (node.contentType === 'text/html') {
                    let failedToParseHtml = false;
                    if ((!alternative && this.hasText) || (node.root && !this.hasText)) {
                        if (this.options.skipHtmlToText) {
                            text.push('');
                        } else if (node.textContent.length > this.options.maxHtmlLengthToP
                            this.emit('error', new Error(`HTML too long for parsing ${node
                            text.push('Invalid HTML content (too long)');
                            failedToParseHtml = true;
                        } else {
                            try {
                                text.push(htmlToText(node.textContent));
                            } catch (err) {
                                this.emit('error', new Error('Failed to parse HTML'));
                                text.push('Invalid HTML content');
                                failedToParseHtml = true;
                            }
                        }
                    }
                    if (!failedToParseHtml) {
                        html.push(node.textContent);
                    }
                }
            }
            alternative = alternative || node.contentType === 'multipart/alternative';
            node.children.forEach(subNode => {
                processNode(alternative, level + 1, subNode);
            });
        };

        processNode(false, 0, this.tree);

        let response = {
            type: 'text'
        };
        if (html.length) {
            this.html = response.html = html.join('<br/>\n');
        }
```

```
765         if (text.length) {
766             this.text = response.text = text.join('\n');
767             this.textAsHtml = response.textAsHtml = text.map(part => this.textToHtml(part)
768         }
769         return response;
770     }
771
772     processChunk(data, done) {
773         let partId = null;
774         if (data._parentBoundary) {
775             partId = this._getPartId(data._parentBoundary);
776         }
777         switch (data.type) {
778             case 'node': {
779                 let node = this.createNode(data);
780                 if (node === this.tree) {
781                     ['subject', 'references', 'date', 'to', 'from', 'to', 'cc', 'bcc', 'me
782                         if (node.headers.has(key)) {
783                             this[key.replace(/-([a-z])/g, (m, c) => c.toUpperCase())] = no
784                         }
785                     });
786                     this.emit('headers', node.headers);
787                 }
788
789                 if (data.contentType === 'message/rfc822' && data.messageNode) {
790                     break;
791                 }
792
793                 if (data.parentNode && data.parentNode.contentType === 'message/rfc822') {
794                     node.showMeta = true;
795                 }
796
797                 if (node.isAttachment) {
798                     let contentType = node.contentType;
799                     if (node.contentType === 'application/octet-stream' && data.filename)
800                         contentType = this.libmime.detectMimeType(data.filename) || 'appli
801                     }
802
803                     let attachment = {
804                         type: 'attachment',
805                         content: null,
806                         contentType,
807                         partId,
808                         release: () => {
809                             attachment.release = null;
810                             if (this.waitUntilAttachmentEnd && typeof this.attachmentCallb
811                                 setImmediate(this.attachmentCallback);
812                             }
813                             this.attachmentCallback = false;
```

```
                        this.waitUntilAttachmentEnd = false;
                    }
                };

                let algo = this.options.checksumAlgo || 'md5';
                let hasher = new StreamHash(attachment, algo);
                node.decoder.on('error', err => {
                    hasher.emit('error', err);
                });

                node.decoder.on('readable', () => {
                    let chunk;

                    while ((chunk = node.decoder.read()) !== null) {
                        hasher.write(chunk);
                    }
                });

                node.decoder.once('end', () => {
                    hasher.end();
                });

                //node.decoder.pipe(hasher);
                attachment.content = hasher;

                this.waitUntilAttachmentEnd = true;
                if (data.disposition) {
                    attachment.contentDisposition = data.disposition;
                }

                if (data.filename) {
                    attachment.filename = data.filename;
                }

                if (node.headers.has('content-id')) {
                    attachment.contentId = [].concat(node.headers.get('content-id') ||
                    attachment.cid = attachment.contentId.trim().replace(/^<|>$/g, '')
                    // check if the attachment is "related" to text content like an em
                    let parentNode = node;
                    while ((parentNode = parentNode.parent)) {
                        if (parentNode.contentType === 'multipart/related') {
                            attachment.related = true;
                        }
                    }
                }

                attachment.headers = node.headers;
                this.push(attachment);
                this.attachmentList.push(attachment);
```

```
863                 } else if (node.disposition === 'inline') {
864                     let chunks = [];
865                     let chunklen = 0;
866                     node.contentStream = node.decoder;
867
868                     if (node.contentType === 'text/plain') {
869                         this.hasText = true;
870                     } else if (node.contentType === 'text/html') {
871                         this.hasHtml = true;
872                     } else if (node.contentType === 'message/delivery-status' && !this.opt
873                         this.hasText = true;
874                     }
875
876                     if (node.node.flowed) {
877                         let contentStream = node.contentStream;
878                         let flowDecoder = new FlowedDecoder({
879                             delSp: node.node.delSp
880                         });
881                         contentStream.on('error', err => {
882                             flowDecoder.emit('error', err);
883                         });
884                         contentStream.pipe(flowDecoder);
885                         node.contentStream = flowDecoder;
886                     }
887
888                     let charset = node.charset || 'utf-8';
889                     //charset = charset || 'windows-1257';
890
891                     if (!['ascii', 'usascii', 'utf8'].includes(charset.toLowerCase().repla
892                         try {
893                             let contentStream = node.contentStream;
894                             let decodeStream = this.decoder.decodeStream(charset);
895                             contentStream.on('error', err => {
896                                 decodeStream.emit('error', err);
897                             });
898                             contentStream.pipe(decodeStream);
899                             node.contentStream = decodeStream;
900                         } catch (E) {
901                             // do not decode charset
902                         }
903                     }
904
905                     node.contentStream.on('readable', () => {
906                         let chunk;
907                         while ((chunk = node.contentStream.read()) !== null) {
908                             if (typeof chunk === 'string') {
909                                 chunk = Buffer.from(chunk);
910                             }
911                             chunks.push(chunk);
```

```
                            chunklen += chunk.length;
                        }
                    });

                    node.contentStream.once('end', () => {
                        node.textContent = Buffer.concat(chunks, chunklen).toString().repl
                    });

                    node.contentStream.once('error', err => {
                        this.emit('error', err);
                    });
                }

                break;
            }

            case 'data':
                if (this.curnode && this.curnode.decoder) {
                    this.curnode.decoder.end();
                }

                if (this.waitUntilAttachmentEnd) {
                    this.attachmentCallback = done;
                    return;
                }

                // multipart message structure
                // this is not related to any specific 'node' block as it includes
                // everything between the end of some node body and between the next heade
                //process.stdout.write(data.value);
                break;

            case 'body':
                if (this.curnode && this.curnode.decoder && this.curnode.decoder.writable)
                    if (this.curnode.decoder.write(data.value) === false) {
                        return this.curnode.decoder.once('drain', done);
                    }
                }

                // Leaf element body. Includes the body for the last 'node' block. You mig
                // have several 'body' calls for a single 'node' block
                //process.stdout.write(data.value);
                break;
        }

        setImmediate(done);
    }

    _getPartId(parentBoundary) {
```

```
        let boundaryIndex = this.boundaries.findIndex(item => item.name === parentBoundary
        if (boundaryIndex === -1) {
            this.boundaries.push({ name: parentBoundary, count: 1 });
            boundaryIndex = this.boundaries.length - 1;
        } else {
            this.boundaries[boundaryIndex].count++;
        }
        let partId = '1';
        for (let i = 0; i <= boundaryIndex; i++) {
            if (i === 0) partId = this.boundaries[i].count.toString();
            else partId += '.' + this.boundaries[i].count.toString();
        }
        return partId;
    }

    getAddressesHTML(value) {
        let formatSingleLevel = addresses =>
            addresses
                .map(address => {
                    let str = '<span class="mp_address_group">';
                    if (address.name) {
                        str += '<span class="mp_address_name">' + he.encode(address.name)
                    }
                    if (address.address) {
                        let link = '<a href="mailto:' + he.encode(address.address) + '" cl
                        if (address.name) {
                            str += ' &lt;' + link + '&gt;';
                        } else {
                            str += link;
                        }
                    }
                    if (address.group) {
                        str += formatSingleLevel(address.group) + ';';
                    }
                    return str + '</span>';
                })
                .join(', ');
        return formatSingleLevel([].concat(value || []));
    }

    getAddressesText(value) {
        let formatSingleLevel = addresses =>
            addresses
                .map(address => {
                    let str = '';
                    if (address.name) {
                        str += address.name + (address.group ? ': ' : '');
                    }
                    if (address.address) {
```

```
                        let link = address.address;
                        if (address.name) {
                            str += ' <' + link + '>';
                        } else {
                            str += link;
                        }
                    }
                    if (address.group) {
                        str += formatSingleLevel(address.group) + ';';
                    }
                    return str;
                })
                .join(', ');
        return formatSingleLevel([].concat(value || []));
    }

    updateImageLinks(replaceCallback, done) {
        if (!this.html) {
            return setImmediate(() => done(null, false));
        }

        let cids = new Map();
        let html = (this.html || '').toString();

        if (this.options.skipImageLinks) {
            return done(null, html);
        }

        html.replace(/\bcid:([^'"\s]{1,256})/g, (match, cid) => {
            for (let i = 0, len = this.attachmentList.length; i < len; i++) {
                if (this.attachmentList[i].cid === cid && /^image\/[\w]+$/i.test(this.atta
                    if (/^image\/[\w]+$/i.test(this.attachmentList[i].contentType)) {
                        cids.set(cid, {
                            attachment: this.attachmentList[i]
                        });
                    }
                    break;
                }
            }
            return match;
        });

        let cidList = [];
        cids.forEach(entry => {
            cidList.push(entry);
        });

        let pos = 0;
        let processNext = () => {
```

```
            if (pos >= cidList.length) {
                html = html.replace(/\bcid:([^'"\s]{1,256})/g, (match, cid) => {
                    if (cids.has(cid) && cids.get(cid).url) {
                        return cids.get(cid).url;
                    }
                    return match;
                });

                return done(null, html);
            }
            let entry = cidList[pos++];
            replaceCallback(entry.attachment, (err, url) => {
                if (err) {
                    return setImmediate(() => done(err));
                }
                entry.url = url;
                setImmediate(processNext);
            });
        };

        setImmediate(processNext);
    }

    textToHtml(str) {
        if (this.options.skipTextToHtml) {
            return '';
        }
        str = (str || '').toString();
        let encoded;

        let linkified = false;
        if (!this.options.skipTextLinks) {
            try {
                if (linkify.pretest(str)) {
                    linkified = true;
                    let links = linkify.match(str) || [];
                    let result = [];
                    let last = 0;

                    links.forEach(link => {
                        if (last < link.index) {
                            let textPart = he
                                // encode special chars
                                .encode(str.slice(last, link.index), {
                                    useNamedReferences: true
                                });
                            result.push(textPart);
                        }

```

```
                        result.push(`<a href="${link.url}">${link.text}</a>`);

                        last = link.lastIndex;
                    });

                    let textPart = he
                        // encode special chars
                        .encode(str.slice(last), {
                            useNamedReferences: true
                        });
                    result.push(textPart);

                    encoded = result.join('');
                }
            } catch (E) {
                // failed, don't linkify
            }
        }

        if (!linkified) {
            encoded = he
                // encode special chars
                .encode(str, {
                    useNamedReferences: true
                });
        }

        let text =
            '<p>' +
            encoded
                .replace(/\r?\n/g, '\n')
                .trim() // normalize line endings
                .replace(/[ \t]+$/gm, '')
                .trim() // trim empty line endings
                .replace(/\n\n+/g, '</p><p>')
                .trim() // insert <p> to multiple linebreaks
                .replace(/\n/g, '<br/>') + // insert <br> to single linebreaks
            '</p>';

        return text;
    }
}

module.exports = MailParser;
```