node-irc Documentation

Release 0.10.0

Edward Jones

Oct 18, 2020

Contents

1 More detailed docs: **1**
1.1 API 1

Index **11**

CHAPTER 1

More detailed docs:

1.1 API

This library provides IRC client functionality

1.1.1 Client

`irc.`**`Client`**(*server*, *nick*[, *options*])

This object is the base of the library. It represents a single nick connected to a single IRC server.

The first argument is the server address to connect to, and the second is the nickname to attempt to use. The third optional argument is an options object with default values:

```
{
    userName: 'nodebot',
    realName: 'nodeJS IRC client',
    password: null,
    port: 6667,
    localAddress: null,
    debug: false,
    showErrors: false,
    channels: [],
    autoConnect: true,
    autoRejoin: false,
    autoRenick: false,
    renickCount: null,
    renickDelay: 60000,
    retryCount: null,
    retryDelay: 2000,
    secure: false,
    selfSigned: false,
    certExpired: false,
    floodProtection: false,
```

(continues on next page)

(continued from previous page)

```
    floodProtectionDelay: 1000,
    sasl: false,
    webirc: {
      pass: '',
      ip: '',
      host: ''
    },
    stripColors: false,
    channelPrefixes: "&#",
    messageSplit: 512,
    encoding: null,
    millisecondsOfSilenceBeforePingSent: 15 * 1000,
    millisecondsBeforePingTimeout: 8 * 1000,
    enableStrictParse: false
}
```

`localAddress` is the address to bind to when connecting.

`debug` will output timestamped messages to the console using `util.log` when certain events are fired. If this is true, it will override `showErrors`.

`showErrors` will output timestamped errors to the console using `util.log`, such as when certain IRC responses are encountered or an attempt to find the message charset fails. If `debug` is true, it will override this.

`channels` is an array of channels to join on connect.

`autoConnect` has the client connect when instantiated. If disabled, you will need to call `connect()` on the client instance:

```
var client = new irc.Client({ autoConnect: false, ... });
client.connect();
```

`autoRejoin` has the client rejoin channels after being kicked.

`autoRenick` has the client attempt to renick to its configured nickname if it can't originally join with it (due to nickname clash). Only takes effect when the client receives a `err_nicknameinuse` message – if disabled after this point, will not cancel the effect. See `cancelAutoRenick` to cancel a current renick attempt. See `renickCount` and `renickDelay` to configure.

`renickCount` is the number of times the client will try to automatically renick (reset each time it connects). It defaults to `null` (meaning infinite retry).

`renickDelay` is the number of milliseconds to wait before retrying to automatically renick. It defaults to 60,000ms (60 seconds).

`retryCount` is the number of times the client will try to automatically reconnect when disconnected from the server. It defaults to `null` (meaning infinite retry).

`retryDelay` is the number of milliseconds to wait before retrying to automatically reconnect when disconnected from the server. It defaults to 2000ms (2 seconds).

`secure` (SSL connection) can be a true value or an object (the kind returned by `crypto.createCredentials()`) specifying the certificate and other details for validation. If you set `selfSigned` to true, it will accept certificates from a non-trusted CA. If you set `certExpired` to true, the bot will accept expired certificates.

`floodProtection` queues all your messages and slowly transmits them to prevent being kicked for flooding. Alternatively, use `Client.activateFloodProtection()` to activate flood protection after instantiating the client.

`floodProtectionDelay` sets the amount of time that the client will wait between sending subsequent messages when `floodProtection` is enabled.

`sasl` enables SASL support. You'll also want to set `nick`, `userName`, and `password` for authentication.

`webirc` is an object that contains WEBIRC credentials (if applicable).

`stripColors` removes mIRC colors (0x03 followed by one or two ASCII numbers for the foreground and background color), as well as ircII "effect" codes (`0x02` bold, `0x1f` underline, `0x16` reverse, `0x0f` reset) from the message before parsing it.

`messageSplit` will split up large messages sent with the `say` method into multiple messages of lengths shorter than `messageSplit` bytes, attempting to split at whitespace where possible.

`encoding` specifies the encoding for the bot to convert messages to. To disable this, leave the value blank or false. Example values are `UTF-8` and `ISO-8859-15`.

`millisecondsOfSilenceBeforePingSent` controls the amount of time the ping timer will wait before sending a ping request.

`millisecondsBeforePingTimeout` controls the amount of time the ping timer will wait after sending a ping request before the bot receives a `pingTimeout` event.

`enableStrictParse` will make the client try to conform more strictly to the RFC 2812 standard for parsing nicknames, preventing eg CJK characters from appearing in them.

`Client.`**`connect`**(*[retryCount[, callback]]*)

Connects to the server. Used when `autoConnect` in the options is set to false, or after a disconnect. Outputs an error to console if there is already an active connection. If `retryCount` is a function, it will be treated as a `callback` (i.e. both arguments to this function are optional).

Arguments

- **`retryCount`** (*`integer`*) – an optional number of times to attempt reconnection
- **`callback`** (*`function`*) – an optional callback to fire upon connection

`Client.`**`disconnect`**(*[message[, callback]]*)

Disconnects from the IRC server. If `message` is a function it will be treated as a `callback` (i.e. both arguments to this function are optional). Outputs an error to console if it is already disconnected or disconnecting.

Arguments

- **`message`** (*`string`*) – an optional message to send when disconnecting
- **`callback`** (*`function`*) – an optional callback

`Client.`**`send`**(*command*, *arg1*, *arg2*, ...)

Sends a raw message to the server. Generally speaking, it's best to use other, more specific methods with priority, unless you know what you're doing.

`Client.`**`join`**(*channelList[, callback]*)

Joins the specified channel.

Arguments

- **`channelList`** (*`string`*) – the channel(s) to join
- **`callback`** (*`function`*) – an optional callback to automatically attach to `join#channelname` for each channel

`channelList` supports multiple channels in a comma-separated string (as in the IRC protocol). The callback is called for each channel, but does not include the `channel` parameter (see the `join#channel` event).

To optionally send a list of keys (channel passwords) associated with each channel, add a space after the list of channels and append the list of channels. For example: `#foo,&bar fubar,foobar` will join the channel `#foo` with the key `fubar` and the channel `&bar` with the key `foobar`.

Passing `'0'` to the `channelList` parameter will send `JOIN 0` to the server. As in the IRC spec, this will cause the client to part from all current channels. In such a case, the callback will not be called; you should instead bind to the `part` event to keep track of the progress made.

`Client.`**`part`**(*channel*[, *message*][, *callback*])
Parts the specified channel.

Arguments

- **`channelList`** (*`string`*) – the channel(s) to part
- **`message`** (*`string`*) – an optional message to send upon leaving the channel
- **`callback`** (*`function`*) – a optional callback to automatically attach to `part#channelname` for each channel

As with `Client.join`, the `channelList` parameter supports multiple channels in a comma-separated string, for each of which the callback will be called.

`Client.`**`say`**(*target*, *message*)
Sends a message to the specified target.

Arguments

- **`target`** (*`string`*) – a nickname or a channel to send the message to
- **`message`** (*`string`*) – the message to send

`Client.`**`action`**(*target*, *message*)
Sends an action to the specified target. Often transmitted with `/me` in IRC clients.

Arguments

- **`target`** (*`string`*) – a nickname or a channel to send the action message to
- **`text`** (*`string`*) – the text of the action to send

`Client.`**`notice`**(*target*, *message*)
Sends a notice to the specified target.

Arguments

- **`target`** (*`string`*) – a nickname or a channel to send the notice to
- **`message`** (*`string`*) – the message to send to the target

`Client.`**`whois`**(*nick*, *callback*)
Request a whois for the specified `nick`.

Arguments

- **`nick`** (*`string`*) – a nickname to request a whois of
- **`callback`** (*`function`*) – a callback to fire when the server sends the response; is passed the same information as in the `whois` event above

`Client.`**`ctcp`**(*target*, *type*, *text*)
Sends a CTCP message to the specified target.

Arguments

- **`target`** (*`string`*) – a nickname or a channel to send the CTCP message to

- **type** (*string*) – the type of the CTCP message; that is, "privmsg" for a PRIVMSG, and anything else for a NOTICE
- **text** (*string*) – the CTCP message to send

Client.**list**([*arg1*, *arg2*, ...])
Request a channel listing from the server. The arguments for this method are fairly server specific, so this method passes them through exactly as specified.

Responses from the server are available through the channellist_start, channellist_item, and channellist events.

Client.**activateFloodProtection**([*interval*])
Activates flood protection manually after instantiation of the client. You can also use the floodProtection option while instantiating the client to enable flood protection then; see also floodProtectionDelay to set the message interval.

This should only be called once per Client instance, not on every connection, and cannot currently be deactivated.

Arguments

- **interval** (*integer*) – an optional configuration for amount of time to wait between messages, defaults to client configuration value

Client.**cancelAutoRenick**()
Cancels the current auto-renick event; see the autoRenick config option for more details. Returns the interval object, if it existed.

Client.**canConvertEncoding**()
Calls the exported function irc.canConvertEncoding().

1.1.2 Events

irc.Client instances are EventEmitters with the following events:

'registered'
```
function (message) { }
```
Emitted when the server sends the initial 001 line, indicating you've connected to the server. See the raw event for details on the message object.

'motd'
```
function (motd) { }
```
Emitted when the server sends the message of the day to clients.

'message'
```
function (nick, to, text, message) { }
```
Emitted when a message is sent. The to parameter can be either a nick (which is most likely this client's nick and represents a private message), or a channel (which represents a message to that channel). See the raw event for details on the message object.

'message#'
```
function (nick, to, text, message) { }
```
Emitted when a message is sent to any channel (i.e. exactly the same as the message event but excluding private messages). See the raw event for details on the message object.

'message#channel'
```
function (nick, text, message) { }
```

Same as the 'message' event, but only emitted for the specified channel. See the `raw` event for details on the `message` object.

'selfMessage'

```
function (to, text) { }
```

Emitted when a message is sent from the client. The `to` parameter is the target of the message, which can be either a nick (in a private message) or a channel (as in a message to that channel)

'notice'

```
function (nick, to, text, message) { }
```

Emitted when a notice is sent. The `to` parameter can be either a nick (most likely this client's nick and so represents a private message), or a channel (which represents a message to that channel). The `nick` parameter is either the sender's nick or `null`, representing that the notice comes from the server. See the `raw` event for details on the `message` object.

'action'

```
function (from, to, text, message) { }
```

Emitted whenever a user performs an action (e.g. `/me waves`). See the `raw` event for details on the `message` object.

'pm'

```
function (nick, text, message) { }
```

Same as the 'message' event, but only emitted when the message is directed to the client. See the `raw` event for details on the `message` object.

'invite'

```
function (channel, from, message) { }
```

Emitted when the client receives an `/invite`. See the `raw` event for details on the `message` object.

'names'

```
function (channel, nicks) { }
```

Emitted when the server sends a list of nicks for a channel (which happens immediately after joining or on request). The nicks object passed to the callback is keyed by nickname, and has values '', '+', or '@' depending on the level of that nick in the channel.

'names#channel'

```
function (nicks) { }
```

Same as the 'names' event, but only emitted for the specified channel.

'topic'

```
function (channel, topic, nick, message) { }
```

Emitted when the server sends the channel topic after joining a channel, or when a user changes the topic on a channel. See the `raw` event for details on the `message` object.

'join'

```
function (channel, nick, message) { }
```

Emitted when a user joins a channel (including when the client itself joins a channel). See the `raw` event for details on the `message` object.

'join#channel'

```
function (nick, message) { }
```

Same as the 'join' event, but only emitted for the specified channel. See the `raw` event for details on the `message` object.

'part'
`function (channel, nick, reason, message) { }`

Emitted when a user parts a channel (including when the client itself parts a channel). See the `raw` event for details on the `message` object.

'part#channel'
`function (nick, reason, message) { }`

Same as the 'part' event, but only emitted for the specified channel. See the `raw` event for details on the `message` object.

'quit'
`function (nick, reason, channels, message) { }`

Emitted when a user disconnects from the IRC server, leaving the specified array of channels. Channels are emitted case-lowered.

See the `raw` event for details on the `message` object.

'kick'
`function (channel, nick, by, reason, message) { }`

Emitted when a user is kicked from a channel. See the `raw` event for details on the `message` object.

'kick#channel'
`function (nick, by, reason, message) { }`

Same as the 'kick' event, but only emitted for the specified channel. See the `raw` event for details on the `message` object.

'kill'
`function (nick, reason, channels, message) { }`

Emitted when a user is killed from the IRC server. The `channels` parameter is an array of channels the killed user was in, those known to the client (that is, the ones the bot was present in). Channels are emitted case-lowered.

See the `raw` event for details on the `message` object.

'nick'
`function (oldnick, newnick, channels, message) { }`

Emitted when a user changes nick, with the channels the user is known to be in. Channels are emitted case-lowered.

See the `raw` event for details on the `message` object.

'+mode'
`function (channel, by, mode, argument, message) { }`

Emitted when a mode is added to a user or channel. The `channel` parameter is the channel which the mode is being set on/in. The `by` parameter is the user setting the mode. The `mode` parameter is the single character mode identifier. If the mode is being set on a user, `argument` is the nick of the user. If the mode is being set on a channel, `argument` is the argument to the mode. If a channel mode doesn't have any arguments, `argument` will be 'undefined'. See the `raw` event for details on the `message` object.

'-mode'
`function (channel, by, mode, argument, message) { }`

Emitted when a mode is removed from a user or channel. The other arguments are as in the `+mode` event.

'whois'
`function (info) { }`

Emitted when the server finishes outputting a WHOIS response. The information should look something like:

```
{
    nick: "Throne",
    user: "throne3d",
    host: "10.0.0.1",
    realname: "Unknown",
    channels: ["@#throne3d", "#blah", "#channel"],
    server: "irc.example.com",
    serverinfo: "Example IRC server",
    operator: "is an IRC Operator"
}
```

'ping'
`function (server) { }`

Emitted when a server PINGs the client. The client will automatically send a PONG request just before this is emitted.

'ctcp'
`function (from, to, text, type, message) { }`

Emitted when a CTCP notice or privmsg was received (`type` is either `notice` or `privmsg`). See the `raw` event for details on the `message` object.

'ctcp-notice'
`function (from, to, text, message) { }`

Emitted when a CTCP notice is received. See the `raw` event for details on the `message` object.

'ctcp-privmsg'
`function (from, to, text, message) { }`

Emitted when a CTCP privmsg was received. See the `raw` event for details on the `message` object.

'ctcp-version'
`function (from, to, message) { }`

Emitted when a CTCP VERSION request is received. See the `raw` event for details on the `message` object.

'channellist_start'
`function () {}`

Emitted when the server starts a new channel listing.

'channellist_item'
`function (channel_info) {}`

Emitted for each channel the server returns in a channel listing. The `channel_info` object contains keys 'name', 'users' (number of users in the channel), and 'topic'.

'channellist'
`function (channel_list) {}`

Emitted when the server has finished returning a channel list. The `channel_list` array is simply a list of the objects that were returned in the intervening `channellist_item` events.

This data is also available through the `Client.channellist` property after this event has fired.

'raw'
`function (message) { }`

Emitted when the client receives a "message" from the server. A message is a single line of data from the server. The `message` parameter to the callback is the processed version of this message, and contains something of the form:

```
message = {
    prefix: "user!~realname@example.host", // the prefix for the message␣
↪(optional, user prefix here)
    prefix: "irc.example.com", // the prefix for the message (optional, server␣
↪prefix here)
    nick: "user", // the nickname portion of the prefix (if the prefix is a user␣
↪prefix)
    user: "~realname", // the username portion of the prefix (if the prefix is a␣
↪user prefix)
    host: "example.host", // the hostname portion of the prefix (if the prefix is␣
↪a user prefix)
    server: "irc.example.com", // the server address (if the prefix was a server␣
↪prefix)
    rawCommand: "PRIVMSG", // the command exactly as sent from the server
    command: "PRIVMSG", // human-readable version of the command (if it was␣
↪previously, say, numeric)
    commandType: "normal", // normal, error, or reply
    args: ['#test', 'test message'] // arguments to the command
}
```

You can read more about the IRC protocol in RFC 1459 and RFC 2812.

'error'

`function (message) { }`

Emitted whenever the server responds with an error-type message. See the `raw` event for details on the `message` object. Unhandled messages, although they are shown as errors in the log, are not emitted using this event: see `unhandled`.

'netError'

`function (exception) { }`

Emitted when the socket connection to the server emits an error event. See net.Socket's error event for more information.

'unhandled'

`function (message) { }`

Emitted whenever the server responds with a message the bot doesn't recognize and doesn't handle. See the `raw` event for details on the `message` object.

This must not be relied on to emit particular event codes, as the codes the bot does and does not handle can change between minor versions. It should instead be used as a handler to do something when the bot does not recognize a message, such as warning a user.

1.1.3 Colors

`irc.colors.`**`wrap`**(*color*, *text*[, *reset_color*])

Takes a color by name, text, and optionally what color to return to after the text.

Arguments

- **`color`** (*`string`*) – the name of the desired color, as a string
- **`text`** (*`string`*) – the text you want colorized

- **reset_color** (*string*) – the name of the color you want set after the text (defaults to 'reset')

`irc.colors.`**codes**
Lists the colors available and the relevant mIRC color codes.

```
{
    white: '\u000300',
    black: '\u000301',
    dark_blue: '\u000302',
    dark_green: '\u000303',
    light_red: '\u000304',
    dark_red: '\u000305',
    magenta: '\u000306',
    orange: '\u000307',
    yellow: '\u000308',
    light_green: '\u000309',
    cyan: '\u000310',
    light_cyan: '\u000311',
    light_blue: '\u000312',
    light_magenta: '\u000313',
    gray: '\u000314',
    light_gray: '\u000315',
    reset: '\u000f',
}
```

1.1.4 Encoding

`irc.`**canConvertEncoding**`()`
Tests if the library can convert messages with different encodings, using the `chardet` and `iconv-lite` libraries. Allows you to more easily (programmatically) detect if the `encoding` option will result in any effect, instead of setting it and otherwise resulting in errors. (See also `Client.canConvertEncoding`, an alias for this function.)

1.1.5 Internal

`Client.`**conn**
Socket to the server. Rarely, if ever, needed; use `Client.send` instead.

`Client.`**chans**
The list of channels joined. Includes channel modes, user lists, and topic information. It is only updated *after* the server recognizes the join.

`Client.`**nick**
The current nick of the client. Updated if the nick changes (e.g. upon nick collision when connecting to a server).

`Client.`**_whoisData**
A buffer of whois data, as whois responses are sent over multiple messages.

`Client.`**_addWhoisData**(*nick*, *key*, *value*, *onlyIfExists*)
Adds the relevant whois data (key-value pair), for the specified nick, optionally only if the value exists (is truthy).

`Client.`**_clearWhoisData**(*nick*)
Clears whois data for the specified nick.

Index

Symbols

'+mode' (*global variable or constant*), 7
'-mode' (*global variable or constant*), 7
'action' (*global variable or constant*), 6
'channellist' (*global variable or constant*), 8
'channellist_item' (*global variable or constant*), 8
'channellist_start' (*global variable or constant*), 8
'ctcp' (*global variable or constant*), 8
'ctcp-notice' (*global variable or constant*), 8
'ctcp-privmsg' (*global variable or constant*), 8
'ctcp-version' (*global variable or constant*), 8
'error' (*global variable or constant*), 9
'invite' (*global variable or constant*), 6
'join' (*global variable or constant*), 6
'join#channel' (*global variable or constant*), 6
'kick' (*global variable or constant*), 7
'kick#channel' (*global variable or constant*), 7
'kill' (*global variable or constant*), 7
'message' (*global variable or constant*), 5
'message#' (*global variable or constant*), 5
'message#channel' (*global variable or constant*), 5
'motd' (*global variable or constant*), 5
'names' (*global variable or constant*), 6
'names#channel' (*global variable or constant*), 6
'netError' (*global variable or constant*), 9
'nick' (*global variable or constant*), 7
'notice' (*global variable or constant*), 6
'part' (*global variable or constant*), 6
'part#channel' (*global variable or constant*), 7
'ping' (*global variable or constant*), 8
'pm' (*global variable or constant*), 6
'quit' (*global variable or constant*), 7
'raw' (*global variable or constant*), 8
'registered' (*global variable or constant*), 5
'selfMessage' (*global variable or constant*), 6
'topic' (*global variable or constant*), 6
'unhandled' (*global variable or constant*), 9
'whois' (*global variable or constant*), 7

C

Client._addWhoisData() (*Client method*), 10
Client._clearWhoisData() (*Client method*), 10
Client._whoisData (*global variable or constant*), 10
Client.action() (*Client method*), 4
Client.activateFloodProtection() (*Client method*), 5
Client.cancelAutoRenick() (*Client method*), 5
Client.canConvertEncoding() (*Client method*), 5
Client.chans (*global variable or constant*), 10
Client.conn (*global variable or constant*), 10
Client.connect() (*Client method*), 3
Client.ctcp() (*Client method*), 4
Client.disconnect() (*Client method*), 3
Client.join() (*Client method*), 3
Client.list() (*Client method*), 5
Client.nick (*global variable or constant*), 10
Client.notice() (*Client method*), 4
Client.part() (*Client method*), 4
Client.say() (*Client method*), 4
Client.send() (*Client method*), 3
Client.whois() (*Client method*), 4

I

irc.canConvertEncoding() (*irc method*), 10
irc.Client() (*irc method*), 1
irc.colors.codes (*global variable or constant*), 10
irc.colors.wrap() (*irc.colors method*), 9