Class: Client

Extends: `undici.Dispatcher`

A basic HTTP/1.1 client, mapped on top a single TCP/TLS connection. Pipelining is disabled by default.

Requests are not guaranteed to be dispatched in order of invocation.

`new Client(url[, options])`

Arguments:

- **url** `URL | string` - Should only include the **protocol, hostname, and port**.
- **options** `ClientOptions` (optional)

Returns: `Client`

Parameter: `ClientOptions`

- **bodyTimeout** `number | null` (optional) - Default: `30e3` - The timeout after which a request will time out, in milliseconds. Monitors time between receiving body data. Use `0` to disable it entirely. Defaults to 30 seconds.
- **headersTimeout** `number | null` (optional) - Default: `30e3` - The amount of time the parser will wait to receive the complete HTTP headers. Defaults to 30 seconds.
- **keepAliveMaxTimeout** `number | null` (optional) - Default: `600e3` - The maximum allowed `keepAliveTimeout` when overridden by *keep-alive* hints from the server. Defaults to 10 minutes.
- **keepAliveTimeout** `number | null` (optional) - Default: `4e3` - The timeout after which a socket without active requests will time out. Monitors time between activity on a connected socket. This value may be overridden by *keep-alive* hints from the server. See MDN: HTTP - Headers - Keep-Alive directives for more details. Defaults to 4 seconds.
- **keepAliveTimeoutThreshold** `number | null` (optional) - Default: `1e3` - A number subtracted from server *keep-alive* hints when overriding `keepAliveTimeout` to account for timing inaccuracies caused by e.g. transport latency. Defaults to 1 second.
- **maxHeaderSize** `number | null` (optional) - Default: `16384` - The maximum length of request headers in bytes. Defaults to 16KiB.
- **pipelining** `number | null` (optional) - Default: `1` - The amount of concurrent requests to be sent over the single TCP/TLS connection according to RFC7230. Carefully consider your workload and environment before enabling concurrent requests as pipelining may reduce performance if used incorrectly. Pipelining is sensitive to network stack settings as well as head of line blocking caused by e.g. long running requests. Set to `0` to disable keep-alive connections.

- **connect** `ConnectOptions | Function | null` (optional) - Default: `null`.
- **strictContentLength** `Boolean` (optional) - Default: `true` - Whether to treat request content length mismatches as errors. If true, an error is thrown when the request content-length header doesn't match the length of the request body.

Parameter: `ConnectOptions` Every Tls option, see here. Furthermore, the following options can be passed:

- **socketPath** `string | null` (optional) - Default: `null` - An IPC endpoint, either Unix domain socket or Windows named pipe.
- **maxCachedSessions** `number | null` (optional) - Default: `100` - Maximum number of TLS cached sessions. Use 0 to disable TLS session caching. Default: 100.
- **timeout** `number | null` (optional) - Default `10e3`
- **servername** `string | null` (optional)

Example - Basic Client instantiation

This will instantiate the undici Client, but it will not connect to the origin until something is queued. Consider using `client.connect` to prematurely connect to the origin, or just call `client.request`.

```
'use strict'
import { Client } from 'undici'

const client = new Client('http://localhost:3000')
```

Example - Custom connector

This will allow you to perform some additional check on the socket that will be used for the next request.

```
'use strict'
import { Client, buildConnector } from 'undici'

const connector = buildConnector({ rejectUnauthorized: false })
const client = new Client('https://localhost:3000', {
  connect (opts, cb) {
    connector(opts, (err, socket) => {
      if (err) {
        cb(err)
      } else if (/* assertion */) {
        socket.destroy()
        cb(new Error('kaboom'))
      } else {
        cb(null, socket)
```

```
      }
    })
  }
})
```

Instance Methods

`Client.close([callback])`

Implements `Dispatcher.close([callback])`.

`Client.destroy([error, callback])`

Implements `Dispatcher.destroy([error, callback])`.

Waits until socket is closed before invoking the callback (or returning a promise if no callback is provided).

`Client.connect(options[, callback])`

See `Dispatcher.connect(options[, callback])`.

`Client.dispatch(options, handlers)`

Implements `Dispatcher.dispatch(options, handlers)`.

`Client.pipeline(options, handler)`

See `Dispatcher.pipeline(options, handler)`.

`Client.request(options[, callback])`

See `Dispatcher.request(options [, callback])`.

`Client.stream(options, factory[, callback])`

See `Dispatcher.stream(options, factory[, callback])`.

`Client.upgrade(options[, callback])`

See `Dispatcher.upgrade(options[, callback])`.

Instance Properties

`Client.closed`

- `boolean`

`true` after `client.close()` has been called.

`Client.destroyed`

- `boolean`

`true` after `client.destroyed()` has been called or `client.close()` has been called and the client shutdown has completed.

`Client.pipelining`

- `number`

Property to get and set the pipelining factor.

Instance Events

Event: `'connect'`

See Dispatcher Event: `'connect'`.

Parameters:

- **origin** `URL`
- **targets** `Array<Dispatcher>`

Emitted when a socket has been created and connected. The client will connect once `client.size > 0`.

Example - Client connect event

```
import { createServer } from 'http'
import { Client } from 'undici'
import { once } from 'events'

const server = createServer((request, response) => {
  response.end('Hello, World!')
}).listen()

await once(server, 'listening')

const client = new Client(`http://localhost:${server.address().port}`)

client.on('connect', (origin) => {
  console.log(`Connected to ${origin}`) // should print before the request body statement
})

try {
  const { body } = await client.request({
    path: '/',
    method: 'GET'
  })
```

```
  body.setEncoding('utf-8')
  body.on('data', console.log)
  client.close()
  server.close()
} catch (error) {
  console.error(error)
  client.close()
  server.close()
}
```

Event: `'disconnect'`

See Dispatcher Event: `'disconnect'`.

Parameters:

- **origin** `URL`
- **targets** `Array<Dispatcher>`
- **error** `Error`

Emitted when socket has disconnected. The error argument of the event is the error which caused the socket to disconnect. The client will reconnect if or once `client.size > 0`.

Example - Client disconnect event

```
import { createServer } from 'http'
import { Client } from 'undici'
import { once } from 'events'

const server = createServer((request, response) => {
  response.destroy()
}).listen()

await once(server, 'listening')

const client = new Client(`http://localhost:${server.address().port}`)

client.on('disconnect', (origin) => {
  console.log(`Disconnected from ${origin}`)
})

try {
  await client.request({
    path: '/',
    method: 'GET'
  })
} catch (error) {
```

```
  console.error(error.message)
  client.close()
  server.close()
}
```

Event: `'drain'`

Emitted when pipeline is no longer busy.

See Dispatcher Event: `'drain'`.

Example - Client drain event

```
import { createServer } from 'http'
import { Client } from 'undici'
import { once } from 'events'

const server = createServer((request, response) => {
  response.end('Hello, World!')
}).listen()

await once(server, 'listening')

const client = new Client(`http://localhost:${server.address().port}`)

client.on('drain', () => {
  console.log('drain event')
  client.close()
  server.close()
})

const requests = [
  client.request({ path: '/', method: 'GET' }),
  client.request({ path: '/', method: 'GET' }),
  client.request({ path: '/', method: 'GET' })
]

await Promise.all(requests)

console.log('requests completed')
```

Event: `'error'`

Invoked for users errors such as throwing in the `onError` handler.