Dispatcher

Extends: `events.EventEmitter`

Dispatcher is the core API used to dispatch requests.

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

Instance Methods

`Dispatcher.close([callback]): Promise`

Closes the dispatcher and gracefully waits for enqueued requests to complete before resolving.

Arguments:

- **callback** `(error: Error | null, data: null) => void` (optional)

Returns: `void | Promise<null>` - Only returns a `Promise` if no `callback` argument was passed

```
dispatcher.close() // -> Promise
dispatcher.close(() => {}) // -> void
```

Example - Request resolves before Client closes

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

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

await once(server, 'listening')

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

try {
  const { body } = await client.request({
      path: '/',
      method: 'GET'
  })
  body.setEncoding('utf8')
  body.on('data', console.log)
} catch (error) {}

await client.close()
```

```
console.log('Client closed')
server.close()
```

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

Starts two-way communications with the requested resource using HTTP CONNECT.

Arguments:

- **options** `ConnectOptions`
- **callback** `(err: Error | null, data: ConnectData | null) => void` (optional)

Returns: `void | Promise<ConnectData>` - Only returns a `Promise` if no `callback` argument was passed

Parameter: `ConnectOptions`

- **path** `string`
- **headers** `UndiciHeaders` (optional) - Default: `null`
- **signal** `AbortSignal | events.EventEmitter | null` (optional) - Default: `null`
- **opaque** `unknown` (optional) - This argument parameter is passed through to `ConnectData`

Parameter: `ConnectData`

- **statusCode** `number`
- **headers** `http.IncomingHttpHeaders`
- **socket** `stream.Duplex`
- **opaque** `unknown`

Example - Connect request with echo

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

const server = createServer((request, response) => {
  throw Error('should never get here')
}).listen()

server.on('connect', (req, socket, head) => {
  socket.write('HTTP/1.1 200 Connection established\r\n\r\n')

  let data = head.toString()
  socket.on('data', (buf) => {
```

```
    data += buf.toString()
  })

  socket.on('end', () => {
    socket.end(data)
  })
})

await once(server, 'listening')

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

try {
  const { socket } = await client.connect({
    path: '/'
  })
  const wanted = 'Body'
  let data = ''
  socket.on('data', d => { data += d })
  socket.on('end', () => {
    console.log(`Data received: ${data.toString()} | Data wanted: ${wanted}`)
    client.close()
    server.close()
  })
  socket.write(wanted)
  socket.end()
} catch (error) { }
```

`Dispatcher.destroy([error, callback]): Promise`

Destroy the dispatcher abruptly with the given error. All the pending and running requests will be asynchronously aborted and error. Since this operation is asynchronously dispatched there might still be some progress on dispatched requests.

Both arguments are optional; the method can be called in four different ways:

Arguments:

- **error** `Error | null` (optional)
- **callback** `(error: Error | null, data: null) => void` (optional)

Returns: `void | Promise<void>` - Only returns a `Promise` if no `callback` argument was passed

```
dispatcher.destroy() // -> Promise
dispatcher.destroy(new Error()) // -> Promise
dispatcher.destroy(() => {}) // -> void
```

```
dispatcher.destroy(new Error(), () => {}) // -> void
```

Example - Request is aborted when Client is destroyed

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

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

await once(server, 'listening')

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

try {
  const request = client.request({
    path: '/',
    method: 'GET'
  })
  client.destroy()
    .then(() => {
      console.log('Client destroyed')
      server.close()
    })
  await request
} catch (error) {
  console.error(error)
}
```

`Dispatcher.dispatch(options, handler)`

This is the low level API which all the preceding APIs are implemented on top of. This API is expected to evolve through semver-major versions and is less stable than the preceding higher level APIs. It is primarily intended for library developers who implement higher level APIs on top of this.

Arguments:

- **options** `DispatchOptions`
- **handler** `DispatchHandler`

Returns: `Boolean` - `false` if dispatcher is busy and further dispatch calls won't make any progress until the `'drain'` event has been emitted.

Parameter: `DispatchOptions`

- **origin** `string | URL`
- **path** `string`
- **method** `string`
- **body** `string | Buffer | Uint8Array | stream.Readable | Iterable | AsyncIterable | null` (optional) - Default: `null`
- **headers** `UndiciHeaders` (optional) - Default: `null`
- **idempotent** `boolean` (optional) - Default: `true` if `method` is `'HEAD'` or `'GET'` - Whether the requests can be safely retried or not. If `false` the request won't be sent until all preceding requests in the pipeline has completed.
- **blocking** `boolean` (optional) - Default: `false` - Whether the response is expected to take a long time and would end up blocking the pipeline. When this is set to `true` further pipelining will be avoided on the same connection until headers have been received.
- **upgrade** `string | null` (optional) - Default: `null` - Upgrade the request. Should be used to specify the kind of upgrade i.e. `'Websocket'`.
- **bodyTimeout** `number | null` (optional) - 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) - The amount of time the parser will wait to receive the complete HTTP headers. Defaults to 30 seconds.

Parameter: `DispatchHandler`

- **onConnect** `(abort: () => void, context: object) => void` - Invoked before request is dispatched on socket. May be invoked multiple times when a request is retried when the request at the head of the pipeline fails.
- **onError** `(error: Error) => void` - Invoked when an error has occurred. May not throw.
- **onUpgrade** `(statusCode: number, headers: Buffer[], socket: Duplex) => void` (optional) - Invoked when request is upgraded. Required if `DispatchOptions.upgrade` is defined or `DispatchOptions.method === 'CONNECT'`.
- **onHeaders** `(statusCode: number, headers: Buffer[], resume: () => void) => boolean` - Invoked when statusCode and headers have been received. May be invoked multiple times due to 1xx informational headers. Not required for `upgrade` requests.
- **onData** `(chunk: Buffer) => boolean` - Invoked when response payload data is received. Not required for `upgrade` requests.
- **onComplete** `(trailers: Buffer[]) => void` - Invoked when response payload and trailers have been received and the request has completed. Not required for `upgrade` requests.
- **onBodySent** `(chunk: string | Buffer | Uint8Array) => void` - Invoked when a body chunk is sent to the server. Not required. For a stream

or iterable body this will be invoked for every chunk. For other body types, it will be invoked once after the body is sent.

Example 1 - Dispatch GET request

```
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}`)

const data = []

client.dispatch({
  path: '/',
  method: 'GET',
  headers: {
    'x-foo': 'bar'
  }
}, {
  onConnect: () => {
    console.log('Connected!')
  },
  onError: (error) => {
    console.error(error)
  },
  onHeaders: (statusCode, headers) => {
    console.log(`onHeaders | statusCode: ${statusCode} | headers: ${headers}`)
  },
  onData: (chunk) => {
    console.log('onData: chunk received')
    data.push(chunk)
  },
  onComplete: (trailers) => {
    console.log(`onComplete | trailers: ${trailers}`)
    const res = Buffer.concat(data).toString('utf8')
    console.log(`Data: ${res}`)
    client.close()
    server.close()
  }
```

```
})
```

Example 2 - Dispatch Upgrade Request

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

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

await once(server, 'listening')

server.on('upgrade', (request, socket, head) => {
  console.log('Node.js Server - upgrade event')
  socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n')
  socket.write('Upgrade: WebSocket\r\n')
  socket.write('Connection: Upgrade\r\n')
  socket.write('\r\n')
  socket.end()
})

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

client.dispatch({
  path: '/',
  method: 'GET',
  upgrade: 'websocket'
}, {
  onConnect: () => {
    console.log('Undici Client - onConnect')
  },
  onError: (error) => {
    console.log('onError') // shouldn't print
  },
  onUpgrade: (statusCode, headers, socket) => {
    console.log('Undici Client - onUpgrade')
    console.log(`onUpgrade Headers: ${headers}`)
    socket.on('data', buffer => {
      console.log(buffer.toString('utf8'))
    })
    socket.on('end', () => {
      client.close()
      server.close()
    })
```

```
    socket.end()
  }
})
```

Example 3 - Dispatch POST request

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

const server = createServer((request, response) => {
  request.on('data', (data) => {
    console.log(`Request Data: ${data.toString('utf8')}`)
    const body = JSON.parse(data)
    body.message = 'World'
    response.end(JSON.stringify(body))
  })
}).listen()

await once(server, 'listening')

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

const data = []

client.dispatch({
  path: '/',
  method: 'POST',
  headers: {
    'content-type': 'application/json'
  },
  body: JSON.stringify({ message: 'Hello' })
}, {
  onConnect: () => {
    console.log('Connected!')
  },
  onError: (error) => {
    console.error(error)
  },
  onHeaders: (statusCode, headers) => {
    console.log(`onHeaders | statusCode: ${statusCode} | headers: ${headers}`)
  },
  onData: (chunk) => {
    console.log('onData: chunk received')
    data.push(chunk)
  },
```

```
  onComplete: (trailers) => {
    console.log(`onComplete | trailers: ${trailers}`)
    const res = Buffer.concat(data).toString('utf8')
    console.log(`Response Data: ${res}`)
    client.close()
    server.close()
  }
})
```

`Dispatcher.pipeline(options, handler)`

For easy use with stream.pipeline. The `handler` argument should return a `Readable` from which the result will be read. Usually it should just return the `body` argument unless some kind of transformation needs to be performed based on e.g. `headers` or `statusCode`. The `handler` should validate the response and save any required state. If there is an error, it should be thrown. The function returns a `Duplex` which writes to the request and reads from the response.

Arguments:

- **options** `PipelineOptions`
- **handler** `(data: PipelineHandlerData) => stream.Readable`

Returns: `stream.Duplex`

Parameter: PipelineOptions Extends: `RequestOptions`

- **objectMode** `boolean` (optional) - Default: `false` - Set to `true` if the `handler` will return an object stream.

Parameter: PipelineHandlerData

- **statusCode** `number`
- **headers** `IncomingHttpHeaders`
- **opaque** `unknown`
- **body** `stream.Readable`
- **context** `object`
- **onInfo** `({statusCode: number, headers: Record<string, string | string[]>}) => void | null` (optional) - Default: `null` - Callback collecting all the info headers (HTTP 100-199) received.

Example 1 - Pipeline Echo

```
import { Readable, Writable, PassThrough, pipeline } from 'stream'
import { createServer } from 'http'
import { Client } from 'undici'
import { once } from 'events'
```

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

await once(server, 'listening')

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

let res = ''

pipeline(
  new Readable({
    read () {
      this.push(Buffer.from('undici'))
      this.push(null)
    }
  }),
  client.pipeline({
    path: '/',
    method: 'GET'
  }, ({ statusCode, headers, body }) => {
    console.log(`response received ${statusCode}`)
    console.log('headers', headers)
    return pipeline(body, new PassThrough(), () => {})
  }),
  new Writable({
    write (chunk, _, callback) {
      res += chunk.toString()
      callback()
    },
    final (callback) {
      console.log(`Response pipelined to writable: ${res}`)
      callback()
    }
  }),
  error => {
    if (error) {
      console.error(error)
    }

    client.close()
    server.close()
  }
)
```

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

Performs a HTTP request.

Non-idempotent requests will not be pipelined in order to avoid indirect failures.

Idempotent requests will be automatically retried if they fail due to indirect failure from the request at the head of the pipeline. This does not apply to idempotent requests with a stream request body.

All response bodies must always be fully consumed or destroyed.

Arguments:

- **options** `RequestOptions`
- **callback** `(error: Error | null, data: ResponseData) => void` (optional)

Returns: `void | Promise<ResponseData>` - Only returns a `Promise` if no `callback` argument was passed

Parameter: `RequestOptions` Extends: `DispatchOptions`

- **opaque** `unknown` (optional) - Default: `null` - Used for passing through context to `ResponseData`
- **signal** `AbortSignal | events.EventEmitter | null` (optional) - Default: `null`
- **onInfo** `({statusCode: number, headers: Record<string, string | string[]>}) => void | null` (optional) - Default: `null` - Callback collecting all the info headers (HTTP 100-199) received.

The `RequestOptions.method` property should not be value `'CONNECT'`.

Parameter: `ResponseData`

- **statusCode** `number`
- **headers** `http.IncomingHttpHeaders`
- **body** `stream.Readable` which also implements the body mixin from the Fetch Standard.
- **trailers** `Record<string, string>` - This object starts out as empty and will be mutated to contain trailers after `body` has emitted `'end'`.
- **opaque** `unknown`
- **context** `object`

`body` contains the following additional body mixin methods and properties:

- `text()`
- `json()`
- `arrayBuffer()`
- `body`
- `bodyUsed`

`body` can not be consumed twice. For example, calling `text()` after `json()` throws `TypeError`.

`body` contains the following additional extensions:

- `dump({ limit: Integer })`, dump the response by reading up to `limit` bytes without killing the socket (optional) - Default: 262144.

Example 1 - Basic GET Request

```
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}`)

try {
  const { body, headers, statusCode, trailers } = await client.request({
    path: '/',
    method: 'GET'
  })
  console.log(`response received ${statusCode}`)
  console.log('headers', headers)
  body.setEncoding('utf8')
  body.on('data', console.log)
  body.on('end', () => {
    console.log('trailers', trailers)
  })

  client.close()
  server.close()
} catch (error) {
  console.error(error)
}
```

Example 2 - Aborting a request

> Node.js v15+ is required to run this example

```
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}`)
const abortController = new AbortController()

try {
  client.request({
    path: '/',
    method: 'GET',
    signal: abortController.signal
  })
} catch (error) {
  console.error(error) // should print an RequestAbortedError
  client.close()
  server.close()
}

abortController.abort()
```

Alternatively, any `EventEmitter` that emits an `'abort'` event may be used as an abort controller:

```
import { createServer } from 'http'
import { Client } from 'undici'
import EventEmitter, { 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}`)
const ee = new EventEmitter()

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

```
  console.error(error) // should print an RequestAbortedError
  client.close()
  server.close()
}

ee.emit('abort')
```

Destroying the request or response body will have the same effect.

```
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}`)

try {
  const { body } = await client.request({
    path: '/',
    method: 'GET'
  })
  body.destroy()
} catch (error) {
  console.error(error) // should print an RequestAbortedError
  client.close()
  server.close()
}
```

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

A faster version of `Dispatcher.request`. This method expects the second argument `factory` to return a `stream.Writable` stream which the response will be written to. This improves performance by avoiding creating an intermediate `stream.Readable` stream when the user expects to directly pipe the response body to a `stream.Writable` stream.

As demonstrated in Example 1 - Basic GET stream request, it is recommended to use the `option.opaque` property to avoid creating a closure for the `factory` method. This pattern works well with Node.js Web Frameworks such as Fastify. See Example 2 - Stream to Fastify Response for more details.

Arguments:

- **options** `RequestOptions`
- **factory** `(data: StreamFactoryData) => stream.Writable`
- **callback** `(error: Error | null, data: StreamData) => void` (optional)

Returns: `void | Promise<StreamData>` - Only returns a `Promise` if no `callback` argument was passed

Parameter: `StreamFactoryData`

- **statusCode** `number`
- **headers** `http.IncomingHttpHeaders`
- **opaque** `unknown`
- **onInfo** `({statusCode: number, headers: Record<string, string | string[]>}) => void | null` (optional) - Default: `null` - Callback collecting all the info headers (HTTP 100-199) received.

Parameter: `StreamData`

- **opaque** `unknown`
- **trailers** `Record<string, string>`
- **context** `object`

Example 1 - Basic GET stream request

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

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

await once(server, 'listening')

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

const bufs = []

try {
  await client.stream({
    path: '/',
    method: 'GET',
    opaque: { bufs }
  }, ({ statusCode, headers, opaque: { bufs } }) => {
    console.log(`response received ${statusCode}`)
```

```
    console.log('headers', headers)
    return new Writable({
      write (chunk, encoding, callback) {
        bufs.push(chunk)
        callback()
      }
    })
  })

  console.log(Buffer.concat(bufs).toString('utf-8'))

  client.close()
  server.close()
} catch (error) {
  console.error(error)
}
```

Example 2 - Stream to Fastify Response In this example, a (fake) request is made to the fastify server using `fastify.inject()`. This request then executes the fastify route handler which makes a subsequent request to the raw Node.js http server using `undici.dispatcher.stream()`. The fastify response is passed to the `opaque` option so that undici can tap into the underlying writable stream using `response.raw`. This methodology demonstrates how one could use undici and fastify together to create fast-as-possible requests from one backend server to another.

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

const nodeServer = createServer((request, response) => {
  response.end('Hello, World! From Node.js HTTP Server')
}).listen()

await once(nodeServer, 'listening')

console.log('Node Server listening')

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

const fastifyServer = fastify()

fastifyServer.route({
  url: '/',
  method: 'GET',
```

```
  handler: (request, response) => {
    nodeServerUndiciClient.stream({
      path: '/',
      method: 'GET',
      opaque: response
    }, ({ opaque }) => opaque.raw)
  }
})

await fastifyServer.listen()

console.log('Fastify Server listening')

const fastifyServerUndiciClient = new Client(`http://localhost:${fastifyServer.server.addres

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

  console.log(`response received ${statusCode}`)
  body.setEncoding('utf8')
  body.on('data', console.log)

  nodeServerUndiciClient.close()
  fastifyServerUndiciClient.close()
  fastifyServer.close()
  nodeServer.close()
} catch (error) { }
```

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

Upgrade to a different protocol. Visit MDN - HTTP - Protocol upgrade mechanism for more details.

Arguments:

- **options** `UpgradeOptions`
- **callback** `(error: Error | null, data: UpgradeData) => void` (optional)

Returns: `void | Promise<UpgradeData>` - Only returns a `Promise` if no `callback` argument was passed

Parameter: `UpgradeOptions`

- **path** `string`
- **method** `string` (optional) - Default: `'GET'`
- **headers** `UndiciHeaders` (optional) - Default: `null`
- **protocol** `string` (optional) - Default: `'Websocket'` - A string of comma separated protocols, in descending preference order.
- **signal** `AbortSignal | EventEmitter | null` (optional) - Default: `null`

Parameter: `UpgradeData`

- **headers** `http.IncomingHeaders`
- **socket** `stream.Duplex`
- **opaque** `unknown`

Example 1 - Basic Upgrade Request

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

const server = createServer((request, response) => {
  response.statusCode = 101
  response.setHeader('connection', 'upgrade')
  response.setHeader('upgrade', request.headers.upgrade)
  response.end()
}).listen()

await once(server, 'listening')

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

try {
  const { headers, socket } = await client.upgrade({
    path: '/',
  })
  socket.on('end', () => {
    console.log(`upgrade: ${headers.upgrade}`) // upgrade: Websocket
    client.close()
    server.close()
  })
  socket.end()
} catch (error) {
  console.error(error)
  client.close()
  server.close()
}
```

Instance Events

Event: `'connect'`

Parameters:

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

Event: `'disconnect'`

Parameters:

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

Event: `'connectionError'`

Parameters:

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

Emitted when dispatcher fails to connect to origin.

Event: `'drain'`

Parameters:

- **origin** `URL`

Emitted when dispatcher is no longer busy.

Parameter: `UndiciHeaders`

- `http.IncomingHttpHeaders | string[] | null`

Header arguments such as `options.headers` in `Client.dispatch` can be specified in two forms; either as an object specified by the `http.IncomingHttpHeaders` type, or an array of strings. An array representation of a header list must have an even length or an `InvalidArgumentError` will be thrown.

Keys are lowercase and values are not modified.

Response headers will derive a `host` from the `url` of the Client instance if no `host` header was previously specified.

Example 1 - Object

```
{
  'content-length': '123',
  'content-type': 'text/plain',
  connection: 'keep-alive',
  host: 'mysite.com',
  accept: '*/*'
}
```

Example 2 - Array

```
[
  'content-length', '123',
  'content-type', 'text/plain',
  'connection', 'keep-alive',
  'host', 'mysite.com',
  'accept', '*/*'
]
```