Products Pricing Documentation Community

npm

Sign Up Sign In

Search packages Search

Have ideas to improve npm? **Join in the discussion! »**

node-fetch DT

2.6.1 • Public • Published 10 months ago

Readme

Explore BETA

0 Dependencies

20,376 Dependents

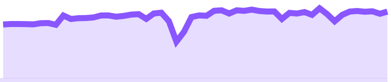
61 Versions

Install

```
> npm i node-fetch
```

Weekly Downloads

20,703,403



Version

2.6.1

License

MIT

Unpacked Size

158 kB

Total Files

8

Issues

105

Pull Requests

25

Homepage

github.com/bitinn/node-fetch

Repository

github.com/bitinn/node-fetch

Last publish

10 months ago

Collaborators

Try on RunKit

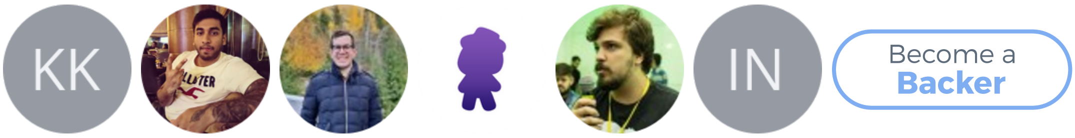
Report malware

node-fetch

npm v2.6.1 travis passing coverage status install size 155 kB Discord 18 online

A light-weight module that brings `window.fetch` to Node.js

(We are looking for v2 maintainers and collaborators)



- Motivation
- Features
- Difference from client-side fetch
- Installation
- Loading and configuring the module
- Common Usage
    - Plain text or HTML

Motivation

Instead of implementing `XMLHttpRequest` in Node.js to run browser-specific **Fetch polyfill**, why not go from native `http` to `fetch` API directly? Hence, `node-fetch`, minimal code for a `window.fetch` compatible API on Node.js runtime.

See Matt Andrews' **isomorphic-fetch** or Leonardo Quixada's **cross-fetch** for isomorphic usage (exports `node-fetch` for server-side, `whatwg-fetch` for client-side).

Features

- Stay consistent with `window.fetch` API.
- Make conscious trade-off when following WHATWG fetch spec and stream spec implementation details, document known differences.
- Use native promise but allow substituting it with [insert your favorite promise library].
- Use native Node streams for body on both request and response.
- Decode content encoding (gzip/deflate) properly and convert string output (such as `res.text()` and `res.json()`) to UTF-8 automatically.
- Useful extensions such as timeout, redirect limit, response size limit, explicit errors for troubleshooting.

Difference from client-side fetch

- See Known Differences for details.
- If you happen to use a missing feature that `window.fetch` offers, feel free to open an issue.
- Pull requests are welcomed too!

Installation

Current stable release (`2.x`)

```
$ npm install node-fetch
```

Loading and configuring the module

We suggest you load the module via `require` until the stabilization of ES modules in node:

```
const fetch = require('node-fetch');
```

If you are using a Promise library other than native, set it through `fetch.Promise` :

```
const Bluebird = require('bluebird');

fetch.Promise = Bluebird;
```

Common Usage

NOTE: The documentation below is up-to-date with `2.x` releases; see the **`1.x` readme**, **changelog** and **2.x upgrade guide** for the differences.

Plain text or HTML

```
fetch('https://github.com/')
    .then(res => res.text())
    .then(body => console.log(body));
```

JSON

```
fetch('https://api.github.com/users/github')
    .then(res => res.json())
    .then(json => console.log(json));
```

Simple Post

```
fetch('https://httpbin.org/post', { method: 'POST', body: 'a=1
    .then(res => res.json()) // expecting a json response
    .then(json => console.log(json));
```

Post with JSON

```
const body = { a: 1 };

fetch('https://httpbin.org/post', {
        method: 'post',
        body:    JSON.stringify(body),
        headers: { 'Content-Type': 'application/json' },
    })
    .then(res => res.json())
    .then(json => console.log(json));
```

Post with form parameters

`URLSearchParams` is available in Node.js as of v7.5.0. See **official documentation** for more usage methods.

NOTE: The `Content-Type` header is only set automatically to `x-www-form-urlencoded` when an instance of `URLSearchParams` is given as such:

```
const { URLSearchParams } = require('url');

const params = new URLSearchParams();
params.append('a', 1);

fetch('https://httpbin.org/post', { method: 'POST', body: para
    .then(res => res.json())
    .then(json => console.log(json));
```

Handling exceptions

NOTE: 3xx-5xx responses are *NOT* exceptions and should be handled in `then()`; see the next section for more information.

Adding a catch to the fetch promise chain will catch *all* exceptions, such as errors originating from node core libraries, network errors and operational errors, which are instances of FetchError. See the **error handling document** for more details.

```
fetch('https://domain.invalid/')
    .catch(err => console.error(err));
```

Handling client and server errors

It is common to create a helper function to check that the response contains no client (4xx) or server (5xx) error responses:

```
function checkStatus(res) {
    if (res.ok) { // res.status >= 200 && res.status < 300
        return res;

    } else {
        throw new MyCustomError(res.statusText);
        throw MyCustomError(res.statusText);
    }
}

fetch('https://httpbin.org/status/400')
    .then(checkStatus)
    .then(res => console.log('will not get here...'))
```

Advanced Usage

Streams

The "Node.js way" is to use streams when possible:

```
fetch('https://assets-cdn.github.com/images/modules/logos_page
    .then(res => {
        const dest = fs.createWriteStream('./octocat.png');
        res.body.pipe(dest);
    });
```

Buffer

If you prefer to cache binary data in full, use buffer(). (NOTE: `buffer()` is a `node-fetch` -only API)

```
const fileType = require('file-type');

fetch('https://assets-cdn.github.com/images/modules/logos_page
    .then(res => res.buffer())
    .then(buffer => fileType(buffer))
    .then(type => { /* ... */ });
```

Accessing Headers and other Meta data

```
fetch('https://github.com/')

    .then(res => {
        console.log(res.ok);
        console.log(res.status);
        console.log(res.statusText);
        console.log(res.headers.raw());
        console.log(res.headers.get('content-type'));
    });
```

Extract Set-Cookie Header

Unlike browsers, you can access raw `Set-Cookie` headers manually using `Headers.raw()`. This is a `node-fetch` only API.

```
fetch(url).then(res => {
    // returns an array of values, instead of a string of com
    console.log(res.headers.raw()['set-cookie']);
});
```

Post data using a file stream

```
const { createReadStream } = require('fs');

const stream = createReadStream('input.txt');

fetch('https://httpbin.org/post', { method: 'POST', body: stre
    .then(res => res.json())
    .then(json => console.log(json));
```

Post with form-data (detect multipart)

```
const FormData = require('form-data');

const form = new FormData();
form.append('a', 1);

fetch('https://httpbin.org/post', { method: 'POST', body: for

    .then(res => res.json())
    .then(json => console.log(json));
    .then(json => console.log(json));

// OR, using custom headers
// NOTE: getHeaders() is non-standard API

const form = new FormData();
form.append('a', 1);

const options = {
    method: 'POST',
    body: form,
    headers: form.getHeaders()
}

fetch('https://httpbin.org/post', options)
    .then(res => res.json())
    .then(json => console.log(json));
```

Request cancellation with AbortSignal

> NOTE: You may cancel streamed requests only on Node >= v8.0.0

You may cancel requests with `AbortController`. A suggested implementation is `abort-controller`.

An example of timing out a request after 150ms could be achieved as the following:

```
import AbortController from 'abort-controller';

const controller = new AbortController();
const timeout = setTimeout(
  () => { controller.abort(); },
  150,
);


fetch(url, { signal: controller.signal })
```

```
    .then(res => res.json())
    .then(
      data => {
        useData(data)
      },
      err => {
        if (err.name === 'AbortError') {
          // request was aborted
        }
      },
    )
    .finally(() => {
      clearTimeout(timeout);
    });
```

See **test cases** for more examples.

API

fetch(url[, options])

- `url` A string representing the URL for fetching
- `options` Options for the HTTP(S) request
- Returns: `Promise<Response>`

Perform an HTTP(S) fetch.

`url` should be an absolute url, such as `https://example.com/`. A path-relative URL (`/file/under/root`) or protocol-relative URL (`//can-be-http-or-https.com/`) will result in a rejected `Promise`.

Options

The default values are shown after each option key.

```
{
    // These properties are part of the Fetch Standard
    method: 'GET',

    headers: {},        // request headers. format is the ide
    body: null,          // request body. can be null, a strin
    redirect: 'follow', // set to `manual` to extract redirec
    signal: null,        // pass an instance of AbortSignal to

    // The following properties are node-fetch extensions
    follow: 20,          // maximum redirect count. 0 to not f
    timeout: 0,          // req/res timeout in ms, it resets o
    compress: true,      // support gzip/deflate content encod
    size: 0,             // maximum response body size in byte
    agent: null          // http(s).Agent instance or function
}
```

Default Headers

If no values are set, the following request headers will be sent automatically:

Header	Value
`Accept-Encoding`	`gzip,deflate` *(when `options.compress === true`)*
`Accept`	`*/*`
`Connection`	`close` *(when no `options.agent` is present)*
`Content-Length`	*(automatically calculated, if possible)*
`Transfer-Encoding`	`chunked` *(when `req.body` is a stream)*
`User-Agent`	`node-fetch/1.0 (+https://github.com/bitinn/node-fetch)`

Note: when `body` is a `Stream`, `Content-Length` is not set automatically.

Custom Agent

The `agent` option allows you to specify networking related options which are out of the scope of Fetch, including and not limited to the following:

- Support self-signed certificate

- Use only IPv4 or IPv6
- Custom DNS Lookup

See `http.Agent` for more information.

In addition, the `agent` option accepts a function that returns `http` (s) `.Agent` instance given current **URL**, this is useful during a redirection chain across HTTP and HTTPS protocol.

```
const httpAgent = new http.Agent({
    keepAlive: true
});
const httpsAgent = new https.Agent({
    keepAlive: true
});

const options = {
    agent: function (_parsedURL) {
        if (_parsedURL.protocol == 'http:') {
            return httpAgent;
        } else {
            return httpsAgent;
        }
    }
}
```

Class: Request

An HTTP(S) request containing information about URL, method, headers, and the body. This class implements the **Body** interface.

Due to the nature of Node.js, the following properties are not implemented at this moment:

- `type`
- `destination`
- `referrer`
- `referrerPolicy`
- `mode`

- `credentials`
- `cache`
- `integrity`
- `keepalive`

The following node-fetch extension properties are provided:

- `follow`
- `compress`
- `counter`
- `agent`

See **options** for exact meaning of these extensions.

new Request(input[, options])

(spec-compliant)

- `input` A string representing a URL, or another `Request` (which will be cloned)
- `options` [Options][#fetch-options] for the HTTP(S) request

Constructs a new `Request` object. The constructor is identical to that in the **browser**.

In most cases, directly `fetch(url, options)` is simpler than creating a `Request` object.

Class: Response

An HTTP(S) response. This class implements the **Body** interface.

The following properties are not implemented in node-fetch at this moment:

- `Response.error()`
- `Response.redirect()`
- `type`
- `trailer`

new Response([body[, options]])

(spec-compliant)

- `body` A `String` or `Readable` stream
- `options` A `ResponseInit` options dictionary

Constructs a new `Response` object. The constructor is identical to that in the **browser**.

Because Node.js does not implement service workers (for which this class was designed), one rarely has to construct a `Response` directly.

response.ok

(spec-compliant)

Convenience property representing if the request ended normally. Will evaluate to true if the response status was greater than or equal to 200 but smaller than 300.

response.redirected

(spec-compliant)

Convenience property representing if the request has been redirected at least once. Will evaluate to true if the internal redirect counter is greater than 0.

Class: Headers

This class allows manipulating and iterating over a set of HTTP headers. All methods specified in the **Fetch Standard** are implemented.

new Headers([init])

(spec-compliant)

- `init` Optional argument to pre-fill the `Headers` object

Construct a new `Headers` object. `init` can be either `null` , a `Headers` object, an key-value map object or any iterable object.

```
// Example adapted from https://fetch.spec.whatwg.org/#exampl

const meta = {
  'Content-Type': 'text/xml',
  'Breaking-Bad': '<3'
};
const headers = new Headers(meta);

// The above is equivalent to

const meta = [
  [ 'Content-Type', 'text/xml' ],
  [ 'Content-Type', 'text/xml' ],
  [ 'Breaking-Bad', '<3' ]
];
const headers = new Headers(meta);

// You can in fact use any iterable objects, like a Map or ev
const meta = new Map();
meta.set('Content-Type', 'text/xml');
meta.set('Breaking-Bad', '<3');
const headers = new Headers(meta);
const copyOfHeaders = new Headers(headers);
```

Interface: Body

`Body` is an abstract interface with methods that are applicable to both `Request` and `Response` classes.

The following methods are not yet implemented in node-fetch at this moment:

- `formData()`

body.body

(deviation from spec)

- Node.js `Readable` stream

Data are encapsulated in the `Body` object. Note that while the **Fetch Standard** requires the property to always be a WHATWG `ReadableStream`, in node-fetch it is a Node.js **`Readable` stream**.

body.bodyUsed

(spec-compliant)

- `Boolean`

A boolean property for if this body has been consumed. Per the specs, a consumed body cannot be used again.

body.arrayBuffer()

body.blob()

body.json()

body.text()

(spec-compliant)

- Returns: `Promise`

Consume the body and return a promise that will resolve to one of these formats.

body.buffer()

(node-fetch extension)

- Returns: `Promise<Buffer>`

Consume the body and return a promise that will resolve to a Buffer.

body.textConverted()

(node-fetch extension)

- Returns: `Promise<String>`

Identical to `body.text()`, except instead of always converting to UTF-8, encoding sniffing will be performed and text converted to UTF-8 if possible.

(This API requires an optional dependency of the npm package **encoding**, which you need to install manually. `webpack` users may see **a warning message** due to this optional dependency.)

Class: FetchError

(node-fetch extension)

An operational error in the fetching process. See **ERROR-HANDLING.md** for more info.

Class: AbortError

(node-fetch extension)

An Error thrown when the request is aborted in response to an `AbortSignal`'s `abort` event. It has a `name` property of `AbortError`. See **ERROR-HANDLING.MD** for more info.

Acknowledgement

Thanks to **github/fetch** for providing a solid implementation reference.

`node-fetch` v1 was maintained by @bitinn; v2 was maintained by @TimothyGu, @bitinn and @jimmywarting; v2 readme is written by @jkantr.

License

MIT

Keywords

fetch http promise

Support

Help

Community

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy