### **🥇 Gold sponsors**

| **Stytch**  API-first authentication, authorization, and fraud prevention  [**Website**](https://stytch.com/?utm_source=oss-sponsorship&utm_medium=paid_sponsorship&utm_content=website-link&utm_campaign=axios-http) | [**Documentation**](https://stytch.com/docs?utm_source=oss-sponsorship&utm_medium=paid_sponsorship&utm_content=docs-link&utm_campaign=axios-http) | [**Node.js**](https://github.com/stytchauth/stytch-node?utm_source=oss-sponsorship&utm_medium=paid_sponsorship&utm_content=node-sdk&utm_campaign=axios-http) | Principal Financial Group  We’re bound by one common purpose: to give you the financial tools, resources and information you ne...  [**www.principal.com**](https://www.principal.com/about-us?utm_source=axios&utm_medium=readme_sponsorlist&utm_campaign=sponsorship) | Descope  Hi, we're Descope! We are building something in the authentication space for app developers and...  [**Website**](https://www.descope.com/?utm_source=axios&utm_medium=referral&utm_campaign=axios-oss-sponsorship) | [**Documentation**](https://docs.descope.com/?utm_source=axios&utm_medium=referral&utm_campaign=axios-oss-sponsorship) | [**Community**](https://www.descope.com/community?utm_source=axios&utm_medium=referral&utm_campaign=axios-oss-sponsorship) |
| --- | --- | --- |
| Buzzoid - Buy Instagram Followers  At Buzzoid, you can buy Instagram followers quickly, safely, and easily with just a few clicks. Rate...  [**buzzoid.com**](https://buzzoid.com/buy-instagram-followers/?utm_source=axios&utm_medium=readme_sponsorlist&utm_campaign=sponsorship) | Famety - Buy Instagram Followers  At Famety, you can grow your social media following quickly, safely, and easily with just a few clic...  [**www.famety.com**](https://www.famety.com/buy-tiktok-likes?utm_source=axios&utm_medium=readme_sponsorlist&utm_campaign=sponsorship) | Poprey - Buy Instagram Likes  Buy Instagram Likes  [**poprey.com**](https://poprey.com/?utm_source=axios&utm_medium=readme_sponsorlist&utm_campaign=sponsorship) |
| [💜 Become a sponsor](https://opencollective.com/axios/contribute) | [💜 Become a sponsor](https://opencollective.com/axios/contribute) | [💜 Become a sponsor](https://opencollective.com/axios/contribute) |



Promise based HTTP client for the browser and node.js

[**Website**](https://axios-http.com/) • [**Documentation**](https://axios-http.com/docs/intro)

[npm version](https://www.npmjs.org/package/axios) [CDNJS](https://cdnjs.com/libraries/axios) [Build status](https://github.com/axios/axios/actions/workflows/ci.yml) [Gitpod Ready-to-Code](https://gitpod.io/#https://github.com/axios/axios) [code coverage](https://coveralls.io/r/mzabriskie/axios) [install size](https://packagephobia.now.sh/result?p=axios) [npm bundle size](https://bundlephobia.com/package/axios@latest) [npm downloads](https://npm-stat.com/charts.html?package=axios) [gitter chat](https://gitter.im/mzabriskie/axios) [code helpers](https://www.codetriage.com/axios/axios) [Known Vulnerabilities](https://snyk.io/test/npm/axios)

## Table of Contents

* [Features](#features)
* [Browser Support](#browser-support)
* [Installing](#installing)
  + [Package manager](#package-manager)
  + [CDN](#cdn)
* [Example](#example)
* [Axios API](#axios-api)
* [Request method aliases](#request-method-aliases)
* [Concurrency 👎](#concurrency-deprecated)
* [Creating an instance](#creating-an-instance)
* [Instance methods](#instance-methods)
* [Request Config](#request-config)
* [Response Schema](#response-schema)
* [Config Defaults](#config-defaults)
  + [Global axios defaults](#global-axios-defaults)
  + [Custom instance defaults](#custom-instance-defaults)
  + [Config order of precedence](#config-order-of-precedence)
* [Interceptors](#interceptors)
  + [Multiple Interceptors](#multiple-interceptors)
* [Handling Errors](#handling-errors)
* [Cancellation](#cancellation)
  + [AbortController](#abortcontroller)
  + [CancelToken 👎](#canceltoken-deprecated)
* [Using application/x-www-form-urlencoded format](#using-applicationx-www-form-urlencoded-format)
  + [URLSearchParams](#urlsearchparams)
  + [Query string](#query-string-older-browsers)
  + [🆕 Automatic serialization](#-automatic-serialization-to-urlsearchparams)
* [Using multipart/form-data format](#using-multipartform-data-format)
  + [FormData](#formdata)
  + [🆕 Automatic serialization](#-automatic-serialization-to-formdata)
* [Files Posting](#files-posting)
* [HTML Form Posting](#-html-form-posting-browser)
* [🆕 Progress capturing](#-progress-capturing)
* [🆕 Rate limiting](#-progress-capturing)
* [🆕 AxiosHeaders](#-axiosheaders)
* [🔥 Fetch adapter](#-fetch-adapter)
* [Semver](#semver)
* [Promises](#promises)
* [TypeScript](#typescript)
* [Resources](#resources)
* [Credits](#credits)
* [License](#license)

## Features

* Make [XMLHttpRequests](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) from the browser
* Make [http](https://nodejs.org/api/http.html) requests from node.js
* Supports the [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) API
* Intercept request and response
* Transform request and response data
* Cancel requests
* Automatic transforms for [JSON](https://www.json.org/json-en.html) data
* 🆕 Automatic data object serialization to multipart/form-data and x-www-form-urlencoded body encodings
* Client side support for protecting against [XSRF](https://en.wikipedia.org/wiki/Cross-site_request_forgery)

## Browser Support

| **Chrome** | **Firefox** | **Safari** | **Opera** | **Edge** |
| --- | --- | --- | --- | --- |
| Latest ✔ | Latest ✔ | Latest ✔ | Latest ✔ | Latest ✔ |

[Browser Matrix](https://saucelabs.com/u/axios)

## Installing

### Package manager

Using npm:

$ npm install axios

Using bower:

$ bower install axios

Using yarn:

$ yarn add axios

Using pnpm:

$ pnpm add axios

Once the package is installed, you can import the library using import or require approach:

import axios, {isCancel, AxiosError} from 'axios';

You can also use the default export, since the named export is just a re-export from the Axios factory:

import axios from 'axios';

console.log(axios.isCancel('something'));

If you use require for importing, **only default export is available**:

const axios = require('axios');

console.log(axios.isCancel('something'));

For some bundlers and some ES6 linter's you may need to do the following:

import { default as axios } from 'axios';

For cases where something went wrong when trying to import a module into a custom or legacy environment, you can try importing the module package directly:

const axios = require('axios/dist/browser/axios.cjs'); // browser commonJS bundle (ES2017)

// const axios = require('axios/dist/node/axios.cjs'); // node commonJS bundle (ES2017)

### CDN

Using jsDelivr CDN (ES5 UMD browser module):

<script src="https://cdn.jsdelivr.net/npm/axios@1.6.7/dist/axios.min.js"></script>

Using unpkg CDN:

<script src="https://unpkg.com/axios@1.6.7/dist/axios.min.js"></script>

## Example

**Note**: CommonJS usage  
In order to gain the TypeScript typings (for intellisense / autocomplete) while using CommonJS imports with require(), use the following approach:

import axios from 'axios';

//const axios = require('axios'); // legacy way

// Make a request for a user with a given ID

axios.get('/user?ID=12345')

.then(function (response) {

// handle success

console.log(response);

})

.catch(function (error) {

// handle error

console.log(error);

})

.finally(function () {

// always executed

});

// Optionally the request above could also be done as

axios.get('/user', {

params: {

ID: 12345

}

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

})

.finally(function () {

// always executed

});

// Want to use async/await? Add the `async` keyword to your outer function/method.

async function getUser() {

try {

const response = await axios.get('/user?ID=12345');

console.log(response);

} catch (error) {

console.error(error);

}

}

**Note**: async/await is part of ECMAScript 2017 and is not supported in Internet Explorer and older browsers, so use with caution.

Performing a POST request

axios.post('/user', {

firstName: 'Fred',

lastName: 'Flintstone'

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

Performing multiple concurrent requests

function getUserAccount() {

return axios.get('/user/12345');

}

function getUserPermissions() {

return axios.get('/user/12345/permissions');

}

Promise.all([getUserAccount(), getUserPermissions()])

.then(function (results) {

const acct = results[0];

const perm = results[1];

});

## axios API

Requests can be made by passing the relevant config to axios.

##### axios(config)

// Send a POST request

axios({

method: 'post',

url: '/user/12345',

data: {

firstName: 'Fred',

lastName: 'Flintstone'

}

});

// GET request for remote image in node.js

axios({

method: 'get',

url: 'https://bit.ly/2mTM3nY',

responseType: 'stream'

})

.then(function (response) {

response.data.pipe(fs.createWriteStream('ada\_lovelace.jpg'))

});

##### axios(url[, config])

// Send a GET request (default method)

axios('/user/12345');

### Request method aliases

For convenience, aliases have been provided for all common request methods.

##### axios.request(config)

##### axios.get(url[, config])

##### axios.delete(url[, config])

##### axios.head(url[, config])

##### axios.options(url[, config])

##### axios.post(url[, data[, config]])

##### axios.put(url[, data[, config]])

##### axios.patch(url[, data[, config]])

###### NOTE

When using the alias methods url, method, and data properties don't need to be specified in config.

### Concurrency (Deprecated)

Please use Promise.all to replace the below functions.

Helper functions for dealing with concurrent requests.

axios.all(iterable) axios.spread(callback)

### Creating an instance

You can create a new instance of axios with a custom config.

##### axios.create([config])

const instance = axios.create({

baseURL: 'https://some-domain.com/api/',

timeout: 1000,

headers: {'X-Custom-Header': 'foobar'}

});

### Instance methods

The available instance methods are listed below. The specified config will be merged with the instance config.

##### axios#request(config)

##### axios#get(url[, config])

##### axios#delete(url[, config])

##### axios#head(url[, config])

##### axios#options(url[, config])

##### axios#post(url[, data[, config]])

##### axios#put(url[, data[, config]])

##### axios#patch(url[, data[, config]])

##### axios#getUri([config])

## Request Config

These are the available config options for making requests. Only the url is required. Requests will default to GET if method is not specified.

{

// `url` is the server URL that will be used for the request

url: '/user',

// `method` is the request method to be used when making the request

method: 'get', // default

// `baseURL` will be prepended to `url` unless `url` is absolute.

// It can be convenient to set `baseURL` for an instance of axios to pass relative URLs

// to methods of that instance.

baseURL: 'https://some-domain.com/api/',

// `transformRequest` allows changes to the request data before it is sent to the server

// This is only applicable for request methods 'PUT', 'POST', 'PATCH' and 'DELETE'

// The last function in the array must return a string or an instance of Buffer, ArrayBuffer,

// FormData or Stream

// You may modify the headers object.

transformRequest: [function (data, headers) {

// Do whatever you want to transform the data

return data;

}],

// `transformResponse` allows changes to the response data to be made before

// it is passed to then/catch

transformResponse: [function (data) {

// Do whatever you want to transform the data

return data;

}],

// `headers` are custom headers to be sent

headers: {'X-Requested-With': 'XMLHttpRequest'},

// `params` are the URL parameters to be sent with the request

// Must be a plain object or a URLSearchParams object

params: {

ID: 12345

},

// `paramsSerializer` is an optional config that allows you to customize serializing `params`.

paramsSerializer: {

//Custom encoder function which sends key/value pairs in an iterative fashion.

encode?: (param: string): string => { /\* Do custom operations here and return transformed string \*/ },

// Custom serializer function for the entire parameter. Allows user to mimic pre 1.x behaviour.

serialize?: (params: Record<string, any>, options?: ParamsSerializerOptions ),

//Configuration for formatting array indexes in the params.

indexes: false // Three available options: (1) indexes: null (leads to no brackets), (2) (default) indexes: false (leads to empty brackets), (3) indexes: true (leads to brackets with indexes).

},

// `data` is the data to be sent as the request body

// Only applicable for request methods 'PUT', 'POST', 'DELETE , and 'PATCH'

// When no `transformRequest` is set, must be of one of the following types:

// - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams

// - Browser only: FormData, File, Blob

// - Node only: Stream, Buffer, FormData (form-data package)

data: {

firstName: 'Fred'

},

// syntax alternative to send data into the body

// method post

// only the value is sent, not the key

data: 'Country=Brasil&City=Belo Horizonte',

// `timeout` specifies the number of milliseconds before the request times out.

// If the request takes longer than `timeout`, the request will be aborted.

timeout: 1000, // default is `0` (no timeout)

// `withCredentials` indicates whether or not cross-site Access-Control requests

// should be made using credentials

withCredentials: false, // default

// `adapter` allows custom handling of requests which makes testing easier.

// Return a promise and supply a valid response (see lib/adapters/README.md)

adapter: function (config) {

/\* ... \*/

},

// Also, you can set the name of the built-in adapter, or provide an array with their names

// to choose the first available in the environment

adapter: 'xhr' // 'fetch' | 'http' | ['xhr', 'http', 'fetch']

// `auth` indicates that HTTP Basic auth should be used, and supplies credentials.

// This will set an `Authorization` header, overwriting any existing

// `Authorization` custom headers you have set using `headers`.

// Please note that only HTTP Basic auth is configurable through this parameter.

// For Bearer tokens and such, use `Authorization` custom headers instead.

auth: {

username: 'janedoe',

password: 's00pers3cret'

},

// `responseType` indicates the type of data that the server will respond with

// options are: 'arraybuffer', 'document', 'json', 'text', 'stream'

// browser only: 'blob'

responseType: 'json', // default

// `responseEncoding` indicates encoding to use for decoding responses (Node.js only)

// Note: Ignored for `responseType` of 'stream' or client-side requests

// options are: 'ascii', 'ASCII', 'ansi', 'ANSI', 'binary', 'BINARY', 'base64', 'BASE64', 'base64url',

// 'BASE64URL', 'hex', 'HEX', 'latin1', 'LATIN1', 'ucs-2', 'UCS-2', 'ucs2', 'UCS2', 'utf-8', 'UTF-8',

// 'utf8', 'UTF8', 'utf16le', 'UTF16LE'

responseEncoding: 'utf8', // default

// `xsrfCookieName` is the name of the cookie to use as a value for xsrf token

xsrfCookieName: 'XSRF-TOKEN', // default

// `xsrfHeaderName` is the name of the http header that carries the xsrf token value

xsrfHeaderName: 'X-XSRF-TOKEN', // default

// `undefined` (default) - set XSRF header only for the same origin requests

withXSRFToken: boolean | undefined | ((config: InternalAxiosRequestConfig) => boolean | undefined),

// `onUploadProgress` allows handling of progress events for uploads

// browser & node.js

onUploadProgress: function ({loaded, total, progress, bytes, estimated, rate, upload = true}) {

// Do whatever you want with the Axios progress event

},

// `onDownloadProgress` allows handling of progress events for downloads

// browser & node.js

onDownloadProgress: function ({loaded, total, progress, bytes, estimated, rate, download = true}) {

// Do whatever you want with the Axios progress event

},

// `maxContentLength` defines the max size of the http response content in bytes allowed in node.js

maxContentLength: 2000,

// `maxBodyLength` (Node only option) defines the max size of the http request content in bytes allowed

maxBodyLength: 2000,

// `validateStatus` defines whether to resolve or reject the promise for a given

// HTTP response status code. If `validateStatus` returns `true` (or is set to `null`

// or `undefined`), the promise will be resolved; otherwise, the promise will be

// rejected.

validateStatus: function (status) {

return status >= 200 && status < 300; // default

},

// `maxRedirects` defines the maximum number of redirects to follow in node.js.

// If set to 0, no redirects will be followed.

maxRedirects: 21, // default

// `beforeRedirect` defines a function that will be called before redirect.

// Use this to adjust the request options upon redirecting,

// to inspect the latest response headers,

// or to cancel the request by throwing an error

// If maxRedirects is set to 0, `beforeRedirect` is not used.

beforeRedirect: (options, { headers }) => {

if (options.hostname === "example.com") {

options.auth = "user:password";

}

},

// `socketPath` defines a UNIX Socket to be used in node.js.

// e.g. '/var/run/docker.sock' to send requests to the docker daemon.

// Only either `socketPath` or `proxy` can be specified.

// If both are specified, `socketPath` is used.

socketPath: null, // default

// `transport` determines the transport method that will be used to make the request. If defined, it will be used. Otherwise, if `maxRedirects` is 0, the default `http` or `https` library will be used, depending on the protocol specified in `protocol`. Otherwise, the `httpFollow` or `httpsFollow` library will be used, again depending on the protocol, which can handle redirects.

transport: undefined, // default

// `httpAgent` and `httpsAgent` define a custom agent to be used when performing http

// and https requests, respectively, in node.js. This allows options to be added like

// `keepAlive` that are not enabled by default.

httpAgent: new http.Agent({ keepAlive: true }),

httpsAgent: new https.Agent({ keepAlive: true }),

// `proxy` defines the hostname, port, and protocol of the proxy server.

// You can also define your proxy using the conventional `http\_proxy` and

// `https\_proxy` environment variables. If you are using environment variables

// for your proxy configuration, you can also define a `no\_proxy` environment

// variable as a comma-separated list of domains that should not be proxied.

// Use `false` to disable proxies, ignoring environment variables.

// `auth` indicates that HTTP Basic auth should be used to connect to the proxy, and

// supplies credentials.

// This will set an `Proxy-Authorization` header, overwriting any existing

// `Proxy-Authorization` custom headers you have set using `headers`.

// If the proxy server uses HTTPS, then you must set the protocol to `https`.

proxy: {

protocol: 'https',

host: '127.0.0.1',

// hostname: '127.0.0.1' // Takes precedence over 'host' if both are defined

port: 9000,

auth: {

username: 'mikeymike',

password: 'rapunz3l'

}

},

// `cancelToken` specifies a cancel token that can be used to cancel the request

// (see Cancellation section below for details)

cancelToken: new CancelToken(function (cancel) {

}),

// an alternative way to cancel Axios requests using AbortController

signal: new AbortController().signal,

// `decompress` indicates whether or not the response body should be decompressed

// automatically. If set to `true` will also remove the 'content-encoding' header

// from the responses objects of all decompressed responses

// - Node only (XHR cannot turn off decompression)

decompress: true, // default

// `insecureHTTPParser` boolean.

// Indicates where to use an insecure HTTP parser that accepts invalid HTTP headers.

// This may allow interoperability with non-conformant HTTP implementations.

// Using the insecure parser should be avoided.

// see options https://nodejs.org/dist/latest-v12.x/docs/api/http.html#http\_http\_request\_url\_options\_callback

// see also https://nodejs.org/en/blog/vulnerability/february-2020-security-releases/#strict-http-header-parsing-none

insecureHTTPParser: undefined, // default

// transitional options for backward compatibility that may be removed in the newer versions

transitional: {

// silent JSON parsing mode

// `true` - ignore JSON parsing errors and set response.data to null if parsing failed (old behaviour)

// `false` - throw SyntaxError if JSON parsing failed (Note: responseType must be set to 'json')

silentJSONParsing: true, // default value for the current Axios version

// try to parse the response string as JSON even if `responseType` is not 'json'

forcedJSONParsing: true,

// throw ETIMEDOUT error instead of generic ECONNABORTED on request timeouts

clarifyTimeoutError: false,

},

env: {

// The FormData class to be used to automatically serialize the payload into a FormData object

FormData: window?.FormData || global?.FormData

},

formSerializer: {

visitor: (value, key, path, helpers) => {}; // custom visitor function to serialize form values

dots: boolean; // use dots instead of brackets format

metaTokens: boolean; // keep special endings like {} in parameter key

indexes: boolean; // array indexes format null - no brackets, false - empty brackets, true - brackets with indexes

},

// http adapter only (node.js)

maxRate: [

100 \* 1024, // 100KB/s upload limit,

100 \* 1024 // 100KB/s download limit

]

}

## Response Schema

The response for a request contains the following information.

{

// `data` is the response that was provided by the server

data: {},

// `status` is the HTTP status code from the server response

status: 200,

// `statusText` is the HTTP status message from the server response

statusText: 'OK',

// `headers` the HTTP headers that the server responded with

// All header names are lowercase and can be accessed using the bracket notation.

// Example: `response.headers['content-type']`

headers: {},

// `config` is the config that was provided to `axios` for the request

config: {},

// `request` is the request that generated this response

// It is the last ClientRequest instance in node.js (in redirects)

// and an XMLHttpRequest instance in the browser

request: {}

}

When using then, you will receive the response as follows:

axios.get('/user/12345')

.then(function (response) {

console.log(response.data);

console.log(response.status);

console.log(response.statusText);

console.log(response.headers);

console.log(response.config);

});

When using catch, or passing a [rejection callback](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) as second parameter of then, the response will be available through the error object as explained in the [Handling Errors](#handling-errors) section.

## Config Defaults

You can specify config defaults that will be applied to every request.

### Global axios defaults

axios.defaults.baseURL = 'https://api.example.com';

// Important: If axios is used with multiple domains, the AUTH\_TOKEN will be sent to all of them.

// See below for an example using Custom instance defaults instead.

axios.defaults.headers.common['Authorization'] = AUTH\_TOKEN;

axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';

### Custom instance defaults

// Set config defaults when creating the instance

const instance = axios.create({

baseURL: 'https://api.example.com'

});

// Alter defaults after instance has been created

instance.defaults.headers.common['Authorization'] = AUTH\_TOKEN;

### Config order of precedence

Config will be merged with an order of precedence. The order is library defaults found in [lib/defaults.js](https://github.com/axios/axios/blob/master/lib/defaults/index.js#L28), then defaults property of the instance, and finally config argument for the request. The latter will take precedence over the former. Here's an example.

// Create an instance using the config defaults provided by the library

// At this point the timeout config value is `0` as is the default for the library

const instance = axios.create();

// Override timeout default for the library

// Now all requests using this instance will wait 2.5 seconds before timing out

instance.defaults.timeout = 2500;

// Override timeout for this request as it's known to take a long time

instance.get('/longRequest', {

timeout: 5000

});

## Interceptors

You can intercept requests or responses before they are handled by then or catch.

// Add a request interceptor

axios.interceptors.request.use(function (config) {

// Do something before request is sent

return config;

}, function (error) {

// Do something with request error

return Promise.reject(error);

});

// Add a response interceptor

axios.interceptors.response.use(function (response) {

// Any status code that lie within the range of 2xx cause this function to trigger

// Do something with response data

return response;

}, function (error) {

// Any status codes that falls outside the range of 2xx cause this function to trigger

// Do something with response error

return Promise.reject(error);

});

If you need to remove an interceptor later you can.

const myInterceptor = axios.interceptors.request.use(function () {/\*...\*/});

axios.interceptors.request.eject(myInterceptor);

You can also clear all interceptors for requests or responses.

const instance = axios.create();

instance.interceptors.request.use(function () {/\*...\*/});

instance.interceptors.request.clear(); // Removes interceptors from requests

instance.interceptors.response.use(function () {/\*...\*/});

instance.interceptors.response.clear(); // Removes interceptors from responses

You can add interceptors to a custom instance of axios.

const instance = axios.create();

instance.interceptors.request.use(function () {/\*...\*/});

When you add request interceptors, they are presumed to be asynchronous by default. This can cause a delay in the execution of your axios request when the main thread is blocked (a promise is created under the hood for the interceptor and your request gets put on the bottom of the call stack). If your request interceptors are synchronous you can add a flag to the options object that will tell axios to run the code synchronously and avoid any delays in request execution.

axios.interceptors.request.use(function (config) {

config.headers.test = 'I am only a header!';

return config;

}, null, { synchronous: true });

If you want to execute a particular interceptor based on a runtime check, you can add a runWhen function to the options object. The request interceptor will not be executed **if and only if** the return of runWhen is false. The function will be called with the config object (don't forget that you can bind your own arguments to it as well.) This can be handy when you have an asynchronous request interceptor that only needs to run at certain times.

function onGetCall(config) {

return config.method === 'get';

}

axios.interceptors.request.use(function (config) {

config.headers.test = 'special get headers';

return config;

}, null, { runWhen: onGetCall });

**Note:** options parameter(having synchronous and runWhen properties) is only supported for request interceptors at the moment.

### Multiple Interceptors

Given you add multiple response interceptors and when the response was fulfilled

* then each interceptor is executed
* then they are executed in the order they were added
* then only the last interceptor's result is returned
* then every interceptor receives the result of its predecessor
* and when the fulfillment-interceptor throws
  + then the following fulfillment-interceptor is not called
  + then the following rejection-interceptor is called
  + once caught, another following fulfill-interceptor is called again (just like in a promise chain).

Read [the interceptor tests](http://./test/specs/interceptors.spec.js) for seeing all this in code.

## Error Types

There are many different axios error messages that can appear that can provide basic information about the specifics of the error and where opportunities may lie in debugging.

The general structure of axios errors is as follows: | Property | Definition | | -------- | ---------- | | message | A quick summary of the error message and the status it failed with. | | name | This defines where the error originated from. For axios, it will always be an 'AxiosError'. | | stack | Provides the stack trace of the error. | | config | An axios config object with specific instance configurations defined by the user from when the request was made | | code | Represents an axios identified error. The table below lists out specific definitions for internal axios error. | | status | HTTP response status code. See [here](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) for common HTTP response status code meanings.

Below is a list of potential axios identified error | Code | Definition | | -------- | ---------- | | ERR\_BAD\_OPTION\_VALUE | Invalid or unsupported value provided in axios configuration. | | ERR\_BAD\_OPTION | Invalid option provided in axios configuration. | | ECONNABORTED | Request timed out due to exceeding timeout specified in axios configuration. | | ETIMEDOUT | Request timed out due to exceeding default axios timelimit. | | ERR\_NETWORK | Network-related issue. | ERR\_FR\_TOO\_MANY\_REDIRECTS | Request is redirected too many times; exceeds max redirects specified in axios configuration. | ERR\_DEPRECATED | Deprecated feature or method used in axios. | ERR\_BAD\_RESPONSE | Response cannot be parsed properly or is in an unexpected format. | ERR\_BAD\_REQUEST | Requested has unexpected format or missing required parameters. | | ERR\_CANCELED | Feature or method is canceled explicitly by the user.  
| ERR\_NOT\_SUPPORT | Feature or method not supported in the current axios environment. | ERR\_INVALID\_URL | Invalid URL provided for axios request.

## Handling Errors

the default behavior is to reject every response that returns with a status code that falls out of the range of 2xx and treat it as an error.

axios.get('/user/12345')

.catch(function (error) {

if (error.response) {

// The request was made and the server responded with a status code

// that falls out of the range of 2xx

console.log(error.response.data);

console.log(error.response.status);

console.log(error.response.headers);

} else if (error.request) {

// The request was made but no response was received

// `error.request` is an instance of XMLHttpRequest in the browser and an instance of

// http.ClientRequest in node.js

console.log(error.request);

} else {

// Something happened in setting up the request that triggered an Error

console.log('Error', error.message);

}

console.log(error.config);

});

Using the validateStatus config option, you can override the default condition (status >= 200 && status < 300) and define HTTP code(s) that should throw an error.

axios.get('/user/12345', {

validateStatus: function (status) {

return status < 500; // Resolve only if the status code is less than 500

}

})

Using toJSON you get an object with more information about the HTTP error.

axios.get('/user/12345')

.catch(function (error) {

console.log(error.toJSON());

});

## Cancellation

### AbortController

Starting from v0.22.0 Axios supports AbortController to cancel requests in fetch API way:

const controller = new AbortController();

axios.get('/foo/bar', {

signal: controller.signal

}).then(function(response) {

//...

});

// cancel the request

controller.abort()

### CancelToken 👎deprecated

You can also cancel a request using a *CancelToken*.

The axios cancel token API is based on the withdrawn [cancellable promises proposal](https://github.com/tc39/proposal-cancelable-promises).

This API is deprecated since v0.22.0 and shouldn't be used in new projects

You can create a cancel token using the CancelToken.source factory as shown below:

const CancelToken = axios.CancelToken;

const source = CancelToken.source();

axios.get('/user/12345', {

cancelToken: source.token

}).catch(function (thrown) {

if (axios.isCancel(thrown)) {

console.log('Request canceled', thrown.message);

} else {

// handle error

}

});

axios.post('/user/12345', {

name: 'new name'

}, {

cancelToken: source.token

})

// cancel the request (the message parameter is optional)

source.cancel('Operation canceled by the user.');

You can also create a cancel token by passing an executor function to the CancelToken constructor:

const CancelToken = axios.CancelToken;

let cancel;

axios.get('/user/12345', {

cancelToken: new CancelToken(function executor(c) {

// An executor function receives a cancel function as a parameter

cancel = c;

})

});

// cancel the request

cancel();

**Note:** you can cancel several requests with the same cancel token/abort controller. If a cancellation token is already cancelled at the moment of starting an Axios request, then the request is cancelled immediately, without any attempts to make a real request.

During the transition period, you can use both cancellation APIs, even for the same request:

## Using application/x-www-form-urlencoded format

### URLSearchParams

By default, axios serializes JavaScript objects to JSON. To send data in the [application/x-www-form-urlencoded format](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST) instead, you can use the [URLSearchParams](https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams) API, which is [supported](http://www.caniuse.com/#feat=urlsearchparams) in the vast majority of browsers,and  [Node](https://nodejs.org/api/url.html#url_class_urlsearchparams) starting with v10 (released in 2018).

const params = new URLSearchParams({ foo: 'bar' });

params.append('extraparam', 'value');

axios.post('/foo', params);

### Query string (Older browsers)

For compatibility with very old browsers, there is a [polyfill](https://github.com/WebReflection/url-search-params) available (make sure to polyfill the global environment).

Alternatively, you can encode data using the [qs](https://github.com/ljharb/qs) library:

const qs = require('qs');

axios.post('/foo', qs.stringify({ 'bar': 123 }));

Or in another way (ES6),

import qs from 'qs';

const data = { 'bar': 123 };

const options = {

method: 'POST',

headers: { 'content-type': 'application/x-www-form-urlencoded' },

data: qs.stringify(data),

url,

};

axios(options);

### Older Node.js versions

For older Node.js engines, you can use the [querystring](https://nodejs.org/api/querystring.html) module as follows:

const querystring = require('querystring');

axios.post('https://something.com/', querystring.stringify({ foo: 'bar' }));

You can also use the [qs](https://github.com/ljharb/qs) library.

**Note**: The qs library is preferable if you need to stringify nested objects, as the querystring method has [known issues](https://github.com/nodejs/node-v0.x-archive/issues/1665) with that use case.

### 🆕 Automatic serialization to URLSearchParams

Axios will automatically serialize the data object to urlencoded format if the content-type header is set to "application/x-www-form-urlencoded".

const data = {

x: 1,

arr: [1, 2, 3],

arr2: [1, [2], 3],

users: [{name: 'Peter', surname: 'Griffin'}, {name: 'Thomas', surname: 'Anderson'}],

};

await axios.postForm('https://postman-echo.com/post', data,

{headers: {'content-type': 'application/x-www-form-urlencoded'}}

);

The server will handle it as:

{

x: '1',

'arr[]': [ '1', '2', '3' ],

'arr2[0]': '1',

'arr2[1][0]': '2',

'arr2[2]': '3',

'arr3[]': [ '1', '2', '3' ],

'users[0][name]': 'Peter',

'users[0][surname]': 'griffin',

'users[1][name]': 'Thomas',

'users[1][surname]': 'Anderson'

}

If your backend body-parser (like body-parser of express.js) supports nested objects decoding, you will get the same object on the server-side automatically

var app = express();

app.use(bodyParser.urlencoded({ extended: true })); // support encoded bodies

app.post('/', function (req, res, next) {

// echo body as JSON

res.send(JSON.stringify(req.body));

});

server = app.listen(3000);

## Using multipart/form-data format

### FormData

To send the data as a multipart/formdata you need to pass a formData instance as a payload. Setting the Content-Type header is not required as Axios guesses it based on the payload type.

const formData = new FormData();

formData.append('foo', 'bar');

axios.post('https://httpbin.org/post', formData);

In node.js, you can use the [form-data](https://github.com/form-data/form-data) library as follows:

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

const form = new FormData();

form.append('my\_field', 'my value');

form.append('my\_buffer', new Buffer(10));

form.append('my\_file', fs.createReadStream('/foo/bar.jpg'));

axios.post('https://example.com', form)

### 🆕 Automatic serialization to FormData

Starting from v0.27.0, Axios supports automatic object serialization to a FormData object if the request Content-Type header is set to multipart/form-data.

The following request will submit the data in a FormData format (Browser & Node.js):

import axios from 'axios';

axios.post('https://httpbin.org/post', {x: 1}, {

headers: {

'Content-Type': 'multipart/form-data'

}

}).then(({data}) => console.log(data));

In the node.js build, the ([form-data](https://github.com/form-data/form-data)) polyfill is used by default.

You can overload the FormData class by setting the env.FormData config variable, but you probably won't need it in most cases:

const axios = require('axios');

var FormData = require('form-data');

axios.post('https://httpbin.org/post', {x: 1, buf: new Buffer(10)}, {

headers: {

'Content-Type': 'multipart/form-data'

}

}).then(({data}) => console.log(data));

Axios FormData serializer supports some special endings to perform the following operations:

* {} - serialize the value with JSON.stringify
* [] - unwrap the array-like object as separate fields with the same key

**Note**: unwrap/expand operation will be used by default on arrays and FileList objects

FormData serializer supports additional options via config.formSerializer: object property to handle rare cases:

* visitor: Function - user-defined visitor function that will be called recursively to serialize the data object to a FormData object by following custom rules.
* dots: boolean = false - use dot notation instead of brackets to serialize arrays and objects;
* metaTokens: boolean = true - add the special ending (e.g user{}: '{"name": "John"}') in the FormData key. The back-end body-parser could potentially use this meta-information to automatically parse the value as JSON.
* indexes: null|false|true = false - controls how indexes will be added to unwrapped keys of flat array-like objects
  + null - don't add brackets (arr: 1, arr: 2, arr: 3)
  + false(default) - add empty brackets (arr[]: 1, arr[]: 2, arr[]: 3)
  + true - add brackets with indexes (arr[0]: 1, arr[1]: 2, arr[2]: 3)

Let's say we have an object like this one:

const obj = {

x: 1,

arr: [1, 2, 3],

arr2: [1, [2], 3],

users: [{name: 'Peter', surname: 'Griffin'}, {name: 'Thomas', surname: 'Anderson'}],

'obj2{}': [{x:1}]

};

The following steps will be executed by the Axios serializer internally:

const formData = new FormData();

formData.append('x', '1');

formData.append('arr[]', '1');

formData.append('arr[]', '2');

formData.append('arr[]', '3');

formData.append('arr2[0]', '1');

formData.append('arr2[1][0]', '2');

formData.append('arr2[2]', '3');

formData.append('users[0][name]', 'Peter');

formData.append('users[0][surname]', 'Griffin');

formData.append('users[1][name]', 'Thomas');

formData.append('users[1][surname]', 'Anderson');

formData.append('obj2{}', '[{"x":1}]');

Axios supports the following shortcut methods: postForm, putForm, patchForm which are just the corresponding http methods with the Content-Type header preset to multipart/form-data.

## Files Posting

You can easily submit a single file:

await axios.postForm('https://httpbin.org/post', {

'myVar' : 'foo',

'file': document.querySelector('#fileInput').files[0]

});

or multiple files as multipart/form-data:

await axios.postForm('https://httpbin.org/post', {

'files[]': document.querySelector('#fileInput').files

});

FileList object can be passed directly:

await axios.postForm('https://httpbin.org/post', document.querySelector('#fileInput').files)

All files will be sent with the same field names: files[].

## 🆕 HTML Form Posting (browser)

Pass HTML Form element as a payload to submit it as multipart/form-data content.

await axios.postForm('https://httpbin.org/post', document.querySelector('#htmlForm'));

FormData and HTMLForm objects can also be posted as JSON by explicitly setting the Content-Type header to application/json:

await axios.post('https://httpbin.org/post', document.querySelector('#htmlForm'), {

headers: {

'Content-Type': 'application/json'

}

})

For example, the Form

<form id="form">

<input type="text" name="foo" value="1">

<input type="text" name="deep.prop" value="2">

<input type="text" name="deep prop spaced" value="3">

<input type="text" name="baz" value="4">

<input type="text" name="baz" value="5">

<select name="user.age">

<option value="value1">Value 1</option>

<option value="value2" selected>Value 2</option>

<option value="value3">Value 3</option>

</select>

<input type="submit" value="Save">

</form>

will be submitted as the following JSON object:

{

"foo": "1",

"deep": {

"prop": {

"spaced": "3"

}

},

"baz": [

"4",

"5"

],

"user": {

"age": "value2"

}

}

Sending Blobs/Files as JSON (base64) is not currently supported.

## 🆕 Progress capturing

Axios supports both browser and node environments to capture request upload/download progress. The frequency of progress events is forced to be limited to 3 times per second.

await axios.post(url, data, {

onUploadProgress: function (axiosProgressEvent) {

/\*{

loaded: number;

total?: number;

progress?: number; // in range [0..1]

bytes: number; // how many bytes have been transferred since the last trigger (delta)

estimated?: number; // estimated time in seconds

rate?: number; // upload speed in bytes

upload: true; // upload sign

}\*/

},

onDownloadProgress: function (axiosProgressEvent) {

/\*{

loaded: number;

total?: number;

progress?: number;

bytes: number;

estimated?: number;

rate?: number; // download speed in bytes

download: true; // download sign

}\*/

}

});

You can also track stream upload/download progress in node.js:

const {data} = await axios.post(SERVER\_URL, readableStream, {

onUploadProgress: ({progress}) => {

console.log((progress \* 100).toFixed(2));

},

headers: {

'Content-Length': contentLength

},

maxRedirects: 0 // avoid buffering the entire stream

});

**Note:** Capturing FormData upload progress is not currently supported in node.js environments.

**⚠️ Warning** It is recommended to disable redirects by setting maxRedirects: 0 to upload the stream in the **node.js** environment, as follow-redirects package will buffer the entire stream in RAM without following the "backpressure" algorithm.

## 🆕 Rate limiting

Download and upload rate limits can only be set for the http adapter (node.js):

const {data} = await axios.post(LOCAL\_SERVER\_URL, myBuffer, {

onUploadProgress: ({progress, rate}) => {

console.log(`Upload [${(progress\*100).toFixed(2)}%]: ${(rate / 1024).toFixed(2)}KB/s`)

},

maxRate: [100 \* 1024], // 100KB/s limit

});

## 🆕 AxiosHeaders

Axios has its own AxiosHeaders class to manipulate headers using a Map-like API that guarantees caseless work. Although HTTP is case-insensitive in headers, Axios will retain the case of the original header for stylistic reasons and for a workaround when servers mistakenly consider the header's case. The old approach of directly manipulating headers object is still available, but deprecated and not recommended for future usage.

### Working with headers

An AxiosHeaders object instance can contain different types of internal values. that control setting and merging logic. The final headers object with string values is obtained by Axios by calling the toJSON method.

Note: By JSON here we mean an object consisting only of string values intended to be sent over the network.

The header value can be one of the following types:

* string - normal string value that will be sent to the server
* null - skip header when rendering to JSON
* false - skip header when rendering to JSON, additionally indicates that set method must be called with rewrite option set to true to overwrite this value (Axios uses this internally to allow users to opt out of installing certain headers like User-Agent or Content-Type)
* undefined - value is not set

Note: The header value is considered set if it is not equal to undefined.

The headers object is always initialized inside interceptors and transformers:

axios.interceptors.request.use((request: InternalAxiosRequestConfig) => {

request.headers.set('My-header', 'value');

request.headers.set({

"My-set-header1": "my-set-value1",

"My-set-header2": "my-set-value2"

});

request.headers.set('User-Agent', false); // disable subsequent setting the header by Axios

request.headers.setContentType('text/plain');

request.headers['My-set-header2'] = 'newValue' // direct access is deprecated

return request;

}

);

You can iterate over an AxiosHeaders instance using a for...of statement:

const headers = new AxiosHeaders({

foo: '1',

bar: '2',

baz: '3'

});

for(const [header, value] of headers) {

console.log(header, value);

}

// foo 1

// bar 2

// baz 3

### new AxiosHeaders(headers?)

Constructs a new AxiosHeaders instance.

constructor(headers?: RawAxiosHeaders | AxiosHeaders | string);

If the headers object is a string, it will be parsed as RAW HTTP headers.

const headers = new AxiosHeaders(`

Host: www.bing.com

User-Agent: curl/7.54.0

Accept: \*/\*`);

console.log(headers);

// Object [AxiosHeaders] {

// host: 'www.bing.com',

// 'user-agent': 'curl/7.54.0',

// accept: '\*/\*'

// }

### AxiosHeaders#set

set(headerName, value: Axios, rewrite?: boolean);

set(headerName, value, rewrite?: (this: AxiosHeaders, value: string, name: string, headers: RawAxiosHeaders) => boolean);

set(headers?: RawAxiosHeaders | AxiosHeaders | string, rewrite?: boolean);

The rewrite argument controls the overwriting behavior:

* false - do not overwrite if header's value is set (is not undefined)
* undefined (default) - overwrite the header unless its value is set to false
* true - rewrite anyway

The option can also accept a user-defined function that determines whether the value should be overwritten or not.

Returns this.

### AxiosHeaders#get(header)

get(headerName: string, matcher?: true | AxiosHeaderMatcher): AxiosHeaderValue;

get(headerName: string, parser: RegExp): RegExpExecArray | null;

Returns the internal value of the header. It can take an extra argument to parse the header's value with RegExp.exec, matcher function or internal key-value parser.

const headers = new AxiosHeaders({

'Content-Type': 'multipart/form-data; boundary=Asrf456BGe4h'

});

console.log(headers.get('Content-Type'));

// multipart/form-data; boundary=Asrf456BGe4h

console.log(headers.get('Content-Type', true)); // parse key-value pairs from a string separated with \s,;= delimiters:

// [Object: null prototype] {

// 'multipart/form-data': undefined,

// boundary: 'Asrf456BGe4h'

// }

console.log(headers.get('Content-Type', (value, name, headers) => {

return String(value).replace(/a/g, 'ZZZ');

}));

// multipZZZrt/form-dZZZtZZZ; boundZZZry=Asrf456BGe4h

console.log(headers.get('Content-Type', /boundary=(\w+)/)?.[0]);

// boundary=Asrf456BGe4h

Returns the value of the header.

### AxiosHeaders#has(header, matcher?)

has(header: string, matcher?: AxiosHeaderMatcher): boolean;

Returns true if the header is set (has no undefined value).

### AxiosHeaders#delete(header, matcher?)

delete(header: string | string[], matcher?: AxiosHeaderMatcher): boolean;

Returns true if at least one header has been removed.

### AxiosHeaders#clear(matcher?)

clear(matcher?: AxiosHeaderMatcher): boolean;

Removes all headers. Unlike the delete method matcher, this optional matcher will be used to match against the header name rather than the value.

const headers = new AxiosHeaders({

'foo': '1',

'x-foo': '2',

'x-bar': '3',

});

console.log(headers.clear(/^x-/)); // true

console.log(headers.toJSON()); // [Object: null prototype] { foo: '1' }

Returns true if at least one header has been cleared.

### AxiosHeaders#normalize(format);

If the headers object was changed directly, it can have duplicates with the same name but in different cases. This method normalizes the headers object by combining duplicate keys into one. Axios uses this method internally after calling each interceptor. Set format to true for converting headers name to lowercase and capitalize the initial letters (cOntEnt-type => Content-Type)

const headers = new AxiosHeaders({

'foo': '1',

});

headers.Foo = '2';

headers.FOO = '3';

console.log(headers.toJSON()); // [Object: null prototype] { foo: '1', Foo: '2', FOO: '3' }

console.log(headers.normalize().toJSON()); // [Object: null prototype] { foo: '3' }

console.log(headers.normalize(true).toJSON()); // [Object: null prototype] { Foo: '3' }

Returns this.

### AxiosHeaders#concat(...targets)

concat(...targets: Array<AxiosHeaders | RawAxiosHeaders | string | undefined | null>): AxiosHeaders;

Merges the instance with targets into a new AxiosHeaders instance. If the target is a string, it will be parsed as RAW HTTP headers.

Returns a new AxiosHeaders instance.

### AxiosHeaders#toJSON(asStrings?)

toJSON(asStrings?: boolean): RawAxiosHeaders;

Resolve all internal headers values into a new null prototype object. Set asStrings to true to resolve arrays as a string containing all elements, separated by commas.

### AxiosHeaders.from(thing?)

from(thing?: AxiosHeaders | RawAxiosHeaders | string): AxiosHeaders;

Returns a new AxiosHeaders instance created from the raw headers passed in, or simply returns the given headers object if it's an AxiosHeaders instance.

### AxiosHeaders.concat(...targets)

concat(...targets: Array<AxiosHeaders | RawAxiosHeaders | string | undefined | null>): AxiosHeaders;

Returns a new AxiosHeaders instance created by merging the target objects.

### Shortcuts

The following shortcuts are available:

* setContentType, getContentType, hasContentType
* setContentLength, getContentLength, hasContentLength
* setAccept, getAccept, hasAccept
* setUserAgent, getUserAgent, hasUserAgent
* setContentEncoding, getContentEncoding, hasContentEncoding

## 🔥 Fetch adapter

Fetch adapter was introduced in v1.7.0. By default, it will be used if xhr and http adapters are not available in the build, or not supported by the environment. To use it by default, it must be selected explicitly:

const {data} = axios.get(url, {

adapter: 'fetch' // by default ['xhr', 'http', 'fetch']

})

You can create a separate instance for this:

const fetchAxios = axios.create({

adapter: 'fetch'

});

const {data} = fetchAxios.get(url);

The adapter supports the same functionality as xhr adapter, **including upload and download progress capturing**. Also, it supports additional response types such as stream and formdata (if supported by the environment).

## Semver

Until axios reaches a 1.0 release, breaking changes will be released with a new minor version. For example 0.5.1, and 0.5.4 will have the same API, but 0.6.0 will have breaking changes.

## Promises

axios depends on a native ES6 Promise implementation to be [supported](https://caniuse.com/promises). If your environment doesn't support ES6 Promises, you can [polyfill](https://github.com/jakearchibald/es6-promise).

## TypeScript

axios includes [TypeScript](https://typescriptlang.org) definitions and a type guard for axios errors.

let user: User = null;

try {

const { data } = await axios.get('/user?ID=12345');

user = data.userDetails;

} catch (error) {

if (axios.isAxiosError(error)) {

handleAxiosError(error);

} else {

handleUnexpectedError(error);

}

}

Because axios dual publishes with an ESM default export and a CJS module.exports, there are some caveats. The recommended setting is to use "moduleResolution": "node16" (this is implied by "module": "node16"). Note that this requires TypeScript 4.7 or greater. If use ESM, your settings should be fine. If you compile TypeScript to CJS and you can’t use "moduleResolution": "node 16", you have to enable esModuleInterop. If you use TypeScript to type check CJS JavaScript code, your only option is to use "moduleResolution": "node16".

## Online one-click setup

You can use Gitpod, an online IDE(which is free for Open Source) for contributing or running the examples online.

[Open in Gitpod](https://gitpod.io/#https://github.com/axios/axios/blob/main/examples/server.js)

## Resources

* [Changelog](https://github.com/axios/axios/blob/v1.x/CHANGELOG.md)
* [Ecosystem](https://github.com/axios/axios/blob/v1.x/ECOSYSTEM.md)
* [Contributing Guide](https://github.com/axios/axios/blob/v1.x/CONTRIBUTING.md)
* [Code of Conduct](https://github.com/axios/axios/blob/v1.x/CODE_OF_CONDUCT.md)

## Credits

axios is heavily inspired by the [$http service](https://docs.angularjs.org/api/ng/service/$http) provided in [AngularJS](https://angularjs.org/). Ultimately axios is an effort to provide a standalone $http-like service for use outside of AngularJS.

## License

[MIT](http://license)