

# axios

npm v0.27.2 cdnjs v0.27.2 CI passing Gitpod Ready-to-Code coverage 94%

install size 776 kB downloads 137M/month chat on gitter code helpers 149

vulnerabilities 0

Promise based HTTP client for the browser and node.js

New axios docs website: [click here](#)

## Table of Contents







- [Features](#)
- [Browser Support](#)
- [Installing](#)
- [Example](#)
- [Axios API](#)
- [Request method aliases](#)
- [Concurrency](#) 🗨️
- [Creating an instance](#)
- [Instance methods](#)
- [Request Config](#)
- [Response Schema](#)
- [Config Defaults](#)
  - [Global axios defaults](#)
  - [Custom instance defaults](#)
  - [Config order of precedence](#)
- [Interceptors](#)
  - [Multiple Interceptors](#)
- [Handling Errors](#)
- [Cancellation](#)
  - [AbortController](#)
  - [CancelToken](#) 🗨️
- [Using application/x-www-form-urlencoded format](#)
  - [Browser](#)
  - [Node.js](#)
    - [Query string](#)
    - [Form data](#)
      - [Automatic serialization](#)
      - [Manual FormData passing](#)
- [Semver](#)
- [Promises](#)
- [TypeScript](#)
- [Resources](#)

- [Credits](#)
- [License](#)

## Features

- Make [XMLHttpRequests](#) from the browser
- Make [http](#) requests from node.js
- Supports the [Promise](#) API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data
- Client side support for protecting against [XSRF](#)

## Browser Support

					
Latest ✓	Latest ✓	Latest ✓	Latest ✓	Latest ✓	11 ✓



unknown

## Installing

Using npm:

```
$ npm install axios
```

Using bower:

```
$ bower install axios
```

Using yarn:

```
$ yarn add axios
```

Using jsDelivr CDN:

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

Using unpkg CDN:

```
<script src="https://unpkg.com/axios/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:

```
const axios = require('axios').default;

// axios.<method> will now provide autocomplete and parameter typings
```

### Performing a GET request

```
const axios = require('axios').default;

// 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);
  })
  .then(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);
  })
  .then(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: 'http://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 function in charge of serializing `params`
  // (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)
  paramsSerializer: function (params) {
    return Qs.stringify(params, {arrayFormat: 'brackets'})
  },

  // `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
  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) {
    /* ... */
  },

  // `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
  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

  // `onUploadProgress` allows handling of progress events for uploads
  // browser only
  onUploadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
  },

  // `onDownloadProgress` allows handling of progress events for downloads
  // browser only
  onDownloadProgress: function (progressEvent) {
    // Do whatever you want with the native 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

// `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',
  port: 9000,
  auth: {
    username: 'mikeymike',
    password: 'rapunz31'
  }
},

// `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
    }
  }
}

```

## 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 lower cased 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](#) as second parameter of `then`, the response will be available through the `error` object as explained in the [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](#), 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 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 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 });

```

## 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 it's 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](#) for seeing all this in code.

## Handling Errors

```

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 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 [cancelable promises proposal](#).*

*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 real request.*

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

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

By default, axios serializes JavaScript objects to `JSON`. To send data in the `application/x-www-form-urlencoded` format instead, you can use one of the following options.

### Browser

In a browser, you can use the [URLSearchParams](#) API as follows:

```
const params = new URLSearchParams();
params.append('param1', 'value1');
params.append('param2', 'value2');
axios.post('/foo', params);
```

*Note that `URLSearchParams` is not supported by all browsers (see [caniuse.com](#)), but there is a [polyfill](#) available (make sure to polyfill the global environment).*

Alternatively, you can encode data using the [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);
```

## Node.js

### Query string

In node.js, you can use the `querystring` module as follows:

```
const querystring = require('querystring');  
axios.post('http://something.com/', querystring.stringify({ foo: 'bar' }));
```

or `'URLSearchParams'` from `'url module'` as follows:

```
const url = require('url');  
const params = new url.URLSearchParams({ foo: 'bar' });  
axios.post('http://something.com/', params.toString());
```

You can also use the `qs` library.

*NOTE: The `qs` library is preferable if you need to stringify nested objects, as the `querystring` method has [known issues](#) with that use case.*

### Form data

#### Automatic serialization

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`) 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

```

const axios= require('axios');

axios.post('https://httpbin.org/post', {
  'myObj{}': {x: 1, s: "foo"},
  'files[]': document.querySelector('#fileInput').files
}, {
  headers: {
    'Content-Type': 'multipart/form-data'
  }
}).then(({data})=> console.log(data));

```

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

`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[]` ;

### Manual FormData passing

In node.js, you can use the [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)

```

## 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](#). If your environment doesn't support ES6 Promises, you can [polyfill](#).

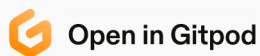
## TypeScript

axios includes [TypeScript](#) 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);
  }
}
```

## Online one-click setup

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



## Resources

- [Changelog](#)
- [Upgrade Guide](#)
- [Ecosystem](#)
- [Contributing Guide](#)
- [Code of Conduct](#)

## Credits

axios is heavily inspired by the [\\$http service](#) provided in [AngularJS](#). Ultimately axios is an effort to provide a standalone `$http`-like service for use outside of AngularJS.

## License

[MIT](#)