**axios**

Клиент для HTTP-запросов для браузера и Node.JS, построенный на [**Promise**](https://learn.javascript.ru/promise)

**Отличительные особенности**

* Отправка [XMLHttpRequests](https://developer.mozilla.org/ru/docs/Web/API/XMLHttpRequest) из браузера
* HTTP запросы из [NodeJS](http://nodejs.org/api/http.html)
* Полная поддержка [Promise](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise) API
* Перехват запросов и ответов
* Преобразование получаемых и принимаемых данных
* Отмена запроса
* Автоматическое преобразование JSON
* Защита на стороне клиента от [CSRF-атак](https://learn.javascript.ru/csrf)

**Browser Support**

| **[Chrome](https://camo.githubusercontent.com/26846e979600799e9f4273d38bd9e5cb7bb8d6d0/68747470733a2f2f7261772e6769746875622e636f6d2f616c7272612f62726f777365722d6c6f676f732f6d61737465722f7372632f6368726f6d652f6368726f6d655f34387834382e706e67)** | **[Firefox](https://camo.githubusercontent.com/6087557f69ec6585eb7f8d7bd7d9ecb6b7f51ba1/68747470733a2f2f7261772e6769746875622e636f6d2f616c7272612f62726f777365722d6c6f676f732f6d61737465722f7372632f66697265666f782f66697265666f785f34387834382e706e67)** | **[Safari](https://camo.githubusercontent.com/6fbaeb334b99e74ddd89190a42766ea3b4600d2c/68747470733a2f2f7261772e6769746875622e636f6d2f616c7272612f62726f777365722d6c6f676f732f6d61737465722f7372632f7361666172692f7361666172695f34387834382e706e67)** | **[Opera](https://camo.githubusercontent.com/96d2405a936da1fb8988db0c1d304d3db04b8a52/68747470733a2f2f7261772e6769746875622e636f6d2f616c7272612f62726f777365722d6c6f676f732f6d61737465722f7372632f6f706572612f6f706572615f34387834382e706e67)** | **[Edge](https://camo.githubusercontent.com/826b3030243b09465bf14cf420704344f5eee991/68747470733a2f2f7261772e6769746875622e636f6d2f616c7272612f62726f777365722d6c6f676f732f6d61737465722f7372632f656467652f656467655f34387834382e706e67)** |  |
| --- | --- | --- | --- | --- | --- |
| Latest ✔ | Latest ✔ | Latest ✔ | Latest ✔ | Latest ✔ | 8+ ✔ |

**Установка**

*посредством****npm****(у вас должен быть установлен пакет Node JS*[*ссылка*](https://nodejs.org/en/)*)*

$ npm install axios

*c помощью менеджера пакетов bower*

$ bower install axios

*используя ссылку на CDN, которую можно разместить непосредственно на странице*

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

**Примеры использования**

**GET - запрос**

// делаем GET запрос чтобы получить пользователя (user)

// с указанным параметром ID = 12345

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

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

// так же, параметры можно передавать отдельно, в виде объекта

// схема ('url', params), где params объект

axios.get('/user', {

params: {

ID: 12345

}

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

// Хотите использовать async/await? Не проблема! Добавьте "async" - перед вашим методом/функуцей,

// и await перед самими запросом.

async function getUser() {

try {

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

console.log(response);

} catch (error) {

console.error(error);

}

}

**FOX:** async/await - часть нового стандарта ECMAScript 2017. Этот функционал не поддерживается IE и некоторыми старыми браузерами. Почитать на русском можно [здесь](https://habr.com/company/ruvds/blog/326074/) Также можно использовать [*BABEL*](https://babeljs.io/) для транспиляции(перевода) кода к стандарту ES5, который имеет практически полную совместимость с браузерами

**POST - запрос**

**FOX** обратите внимание, что зачастую в POST запросе требуется что-то отправлять: объект с параметрами, пустой объект или что-либо еще.

axios.post('/user', {

firstName: 'Fred',

lastName: 'Flintstone'

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

**FOX** в стандарте ES6 вышеуказанный код выглядел бы так(применим стрелочную функцию).

axios.post('/user', {

firstName: 'Fred',

lastName: 'Flintstone'

})

.then(response => console.log(response));

.catch(error => console.log(error));

**FOX** Согласитесь, так выглядит лаконичнее. Более детально о новых фишках в ES6 можно прочесть тут - <https://learn.javascript.ru/es-modern>

**Несколько запросов одновременно**

function getUserAccount() {

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

}

function getUserPermissions() {

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

}

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

.then(axios.spread(function (acct, perms) {

// Оба запроса завершены

}));

**Axios API**

**FOX** Запросы могут быть выполнены путем передачи параметров/настроек в axios(config). То есть вместо стандартноq схемы axios.get('http://yourlink.com/api', parameters) вы указываете каждый параметр(ссылка, объект для передачи, метод и т.д.) вручную. Иногда это может быть удобно, но если Вам нужно просто отправить небольшой запрос и получить данные, воспользуйтесь стандартной схемой(см. раздел GET-запрос)

// отправка POST запроса

axios({

method: 'post',

url: '/user/12345',

data: {

firstName: 'Fred',

lastName: 'Flintstone'

}

});

// отправка GET запроса для получения изображения

// со стороннего сервера

axios({

method:'get',

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

responseType:'stream'

})

.then(function(response) {

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

});

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

// отправка GET запроса с параметрами по умолчанию

axios('/user/12345');

**Встроенные методы HTTP запросов**

Для Вашего удобства были созданы методы для всех поддерживаемых типов запросов.

**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]])**

**FOX** используя данные методы, Вам необязательно указывать свойства method и data в настройках.

**Мультизапросы**

Вспомогательные функции для того чтобы использовать несколько запросов одновременно

**axios.all(iterable)**

**axios.spread(callback)**

**Cоздание нового экземпляра Axios**

Вы можете создать новый экземпляр(образец) **Axios** cо своими настройками

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

const instance = axios.create({

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

timeout: 1000,

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

});

**Методы экземпляра**

Доступные методы экземпляра(образца) Axios перечислены ниже. Указанные настройки в них будут объединены с конфигурацией экземпляра.

**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]])**

**Настройка запросов**

Это параметры для запросов. Если тип запроса не указан, то по умолчанию будет выполняться GET запрос. Запросы указываются при вызове метода, в объекте. *пример: axios.get(params)*, где ***params*** настройки

{

// `url` - адрес сервера. куда отправляется запрос

url: '/user',

// `method` тип http-запроса (get, post, delete и т.д.)

method: 'get', // default

// `baseURL` будет добавлен к `url`, если `url` не будет абсолютным.

// это может быть удобным - установить `baseURL` для экземпляра axios для передачи

// относительных URL-адресов.

// установив единожды в настйроках 'baseURL' не нужно будет указывать полный адрес при запросе.

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

// `transformRequest` позволяет изменять данные запроса до его отправки на сервер

// Это применимо только для методов запроса «PUT», «POST» и «PATCH»

// Последняя функция в массиве должна возвращать строку или экземпляр типа Buffer, ArrayBuffer,

// FormData или Stream

// Также вы можете изменить объект заголовков.

transformRequest: [function (data, headers) {

// тут пишем код для обработки данных...

return data;

}],

// `transformResponse` позволяет изменять данные ответа, (аналог вышеуказанному методу)

// но уже для запроса. редультат передается в then / catch

transformResponse: [function (data) {

// тут пишем код для обработки данных...

return data;

}],

// `headers` - указываем тут заголовки для запроса

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

// `params` - параметры URL, которые обычно указываются в адресе запроса

// параметры должны быть указаны в виде обычного объекта JS

params: {

ID: 12345

},

// `paramsSerializer` - метод для сериализации(обработки) параметров

// (https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)

paramsSerializer: function(params) {

return Qs.stringify(params, {arrayFormat: 'brackets'})

},

// `data` - данные, которые должны быть отправлены в теле запроса

// Применяется только доя методов 'PUT', 'POST', and 'PATCH'

// Если параметр `transformRequest` не установлен, то тело запроса может быть следующих типов:

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

// - Для браузера: FormData, File, Blob

// - Для Node JS: Stream, Buffer

data: {

firstName: 'Fred'

},

// `timeout` количество миллисекунд до истечения времени ожидания запроса.

// Если запрос занимает больше времени, чем `timeout`, он будет прерван.

timeout: 1000,

// `withCredentials` - отображает статус CORS запросов - то есть должны ли запрошиваться

// необходимые парметры или нет. Подробнее тут - https://developer.mozilla.org/ru/docs/Web/HTTP/CORS

withCredentials: false, // default

// `adapter` - позволяет делать доп.настройку запросов, что облегчает тестирование.

// Возвращает Promise и валидный ответ (подробнее в lib/adapters/README.md).

adapter: function (config) {

/\* ... \*/

},

// `auth` указывает, необходимо ли использовать HTTP Basic auth и предоставлять учетные данные.

// ВНИМАНИЕ! Этот параметр установит в запрос заголовок `Authorization`, перезаписав все существующие

// пользовательские заголовки `Authorization`, которые вы задали в настройках

// с помощью параметра - 'headers`

auth: {

username: 'janedoe',

password: 's00pers3cret'

},

// `responseType` указывает тип данных, которыми ответит сервер

// варианты: «arraybuffer», «blob», «document», «json», «text», «stream»,

responseType: 'json', // по умолчанию

// `responseEncoding` указывает какую кодировку использовать для обработки ответов

// Примечание: Игнорируется для опции `responseType` - 'stream'(поток) или запросов на стороне клиента,

// что вполне логично, так как потоковые данные и должны так передаваться

responseEncoding: 'utf8', // default

//`xsrfCookieName` - имя cookie для использования в качестве значения для токена xsrf

xsrfCookieName: 'XSRF-TOKEN', // default

// `xsrfHeaderName` http-заголовко для использования в качестве значения для токена xsrf

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

// `onUploadProgress` позволяет обрабатывать события прогресса для загрузки данных

// например тут можно "повесить" индикатор загружки данных на сервер

onUploadProgress: function (progressEvent) {

// делаем тут что угодно...

},

// `onDownloadProgress` позволяет обрабатывать события прогресса скачивания данных

// как вариант - разместить здесь индикацию размера скачиваемого файла

onDownloadProgress: function (progressEvent) {

// делаем тут что угодно...

},

// `maxContentLength` определяет максимальный размер содержимого ответа HTTP в байтах

maxContentLength: 2000,

// `validateStatus` определяет, разрешать или отклонять Prоmise для данного

// HTTP-ответа. Если `validateStatus` возвращает` true` (или установлен в `null`

// или `undefined`), Promise будет выполнен; в противном случае отклонен

validateStatus: function (status) {

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

},

// `maxRedirects` - максимальное количество редиректов в node.js.

// если значение устанвлено в "0" - редиректа не будет

maxRedirects: 5, // default

// `socketPath` определяет UNIX Socket для использования в node.js.

// например. '/var/run/docker.sock' для отправки запросов демону docker.

// Можно указать только «socketPath» или «proxy».

// Если оба указаны, используется `socketPath`.

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` и` httpsAgent` - установка своего `httpAgent`, который будет использоваться

// при выполнении http и https-запросов. Соответственно, в node.js.

// Это позволяет добавлять опции (например `keepAlive`), которые по умолчанию не включены.

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

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

// 'proxy' определяет имя хоста и порт прокси-сервера

// Используйте `false` для отключения прокси при запросах, игнорируя глобальные среды.

// `auth` указывает, что для подключения к прокси следует использовать HTTP Basic auth и

// предоставляет учетные данные.

// Эта опция установит заголовок `Proxy-Authorization`, переписывая любые существующие

// `Proxy-Authorization` заголовки, которые вы задали в` headers`.

proxy: {

host: '127.0.0.1',

port: 9000,

auth: {

username: 'mikeymike',

password: 'rapunz3l'

}

},

// `cancelToken` указывает токен, который может использоваться для отмены запроса

cancelToken: new CancelToken(function (cancel) {

})

}

**Схема ответа**

Ответ на запрос содержит следующие параметры:

{

// `data` - собственно данные ответа от сервера тут

data: {},

// `status` HTTP код ответа от сервера

// полный список тут - https://developer.mozilla.org/ru/docs/Web/HTTP/Status

status: 200,

// `statusText` - текст сообщение ответа от сервера

statusText: 'OK',

// `headers` - заголовки ответа от сервера.

headers: {},

// `config` - это конфигурация `axios` для запроса

config: {},

// `request` - это запрос, который сгенерировал этот ответ

// Это последний экземпляр ClientRequest в node.js (в переадресации)

// и экземпляр XMLHttpRequest в браузере

request: {}

}

Используя then можно посмотреть ответ таким образом:

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);

});

При использовании catch или передачи [rejection callback](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)(callback для отмены) в качестве 2-го параметра then, ответ будет доступен через объект error, как описано в разделе *Обработка ошибок*

**Настройки по умолчанию**

Вы можете указать настройки по умолчанию, которые будут применяться к каждому запросу.

**Глобальные переменные в Axios**

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

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

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

**Пользовательские настройки для экземплара**

// Установка дефолтных настроек при создании экземпляра

const instance = axios.create({

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

});

// Устанавливаем значения по умолчанию после создания экземпляра

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

**Настройка приоритета**

Конфигурация будет объединена с порядком приоритета. Порядок - это значения по умолчанию в библиотеке, найденные в [lib/defaults.js](https://github.com/axios/axios/blob/master/lib/defaults.js#L28), затем свойство defaults экземпляра и наконец, аргумент config для запроса. Последний будет иметь приоритет над первым. Вот пример

// Создаем образец используя настройки по умолчанию предоставленные библиотекой

// На этом этапе значение конфигурации тайм-аута равно `0`, как по умолчанию для библиотеки

const instance = axios.create();

// Завершение таймаута по умолчанию для библиотеки

// Теперь все запросы с использованием этого экземпляра будут ждать 2,5 секунды до истечения времени ожидания

instance.defaults.timeout = 2500;

// Завершение таймаута для этого запроса, поскольку, он занимает много времени

instance.get('/longRequest', {

timeout: 5000

});

**Перехватчики**

Вы можете перехватить запросы или ответы непосредственно перед тем, как они будут обработаны then или catch.

// Добавление перехвата запроса

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

// делаем что угодно перед запросом

return config;

}, function (error) {

// обрабатываем ошибку

return Promise.reject(error);

});

// Добавляем перехватчик ответа

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

// Делаем что угодно с поступившими данными

return response;

}, function (error) {

// Обрабатываем ошибку

return Promise.reject(error);

});

Если позднее Вам нужно будет удалить перехватчик, Вы можете сделать следующее:

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

axios.interceptors.request.eject(myInterceptor);

Также, Вы можете добавлять перехватчики в свой экземпляр axios

const instance = axios.create();

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

**Обработка ошибок**

axios.get('/user/12345')

.catch(function (error) {

if (error.response) {

// Запрос выполнен, и сервер отправил Вам статус код

// код выпададет из диапазона 2хх (ошибка)

console.log(error.response.data);

console.log(error.response.status);

console.log(error.response.headers);

} else if (error.request) {

// Запрос был сделан, но ответ не получен

// `error.request` - экземпляр XMLHttpRequest в браузере,

// http.ClientRequest экземпляр в node.js

console.log(error.request);

} else {

// Что-то пошло не так, вернулась ошибка

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

}

console.log(error.config);

});

Вы можете определить свой диапазон ошибок кода состояния HTTP, используя опцию конфигурации validateStatus. Например, можно настроить так, что все ошибки с кодом в диапазоне 2хх-3хх будут игонирироваться. В реальности это конечно не пригодится, но возможность такая есть

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

validateStatus: function (status) {

return status < 500; // остановить запрос, только если код больше или равен 500

}

})

**Отмена запроса**

Вы можете отменить запрос, используя *cancel token*.

Token для отмены запросов в Axios базируется на [cancelable promises proposal](https://github.com/tc39/proposal-cancelable-promises).

Вы можете создать token отмены с помощью фабричной функции CancelToken.source, как показано ниже:

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 {

// обработка ошибок

}

});

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

name: 'new name'

}, {

cancelToken: source.token

})

// отмена запроса (отображение сообщения можно настроить дополнительно)

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

Вы также можете создать token для отмены запроса, передав функцию конструктору CancelToken:

const CancelToken = axios.CancelToken;

let cancel;

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

cancelToken: new CancelToken(function executor(c) {

// исполняемая функция получает функцию отмены в качестве параметра

cancel = c;

})

});

// отмена запроса

cancel();

Примечание. Вы можете отменить несколько запросов одним токеном(cancell token).

**Использование формата application/x-www-form-urlencoded**

По умолчанию axios переводит объекты JavaScript в JSON. Чтобы отправить данные в формате application/x-www-form-urlencoded, Вы можете использовать один из следующих вариантов.

**Браузер**

В браузере Вы можете испольприменить API [URLSearchParams](https://developer.mozilla.org/ru/docs/Web/API/URLSearchParams) следующим образом:

const params = new URLSearchParams();

params.append('param1', 'value1');

params.append('param2', 'value2');

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

Обратите внимание, что URLSearchParams поддерживается не всеми браузерами (см. [Caniuse.com](http://www.caniuse.com/#feat=urlsearchparams)), но существует [полифилл](https://github.com/WebReflection/url-search-params)

Кроме того, вы можете декодировать данные, используя библиотеку [qs](https://github.com/ljharb/qs):

const qs = require('qs');

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

Или по-другому... (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**

В Node JS Вы можете использовать модуль [querystring](https://nodejs.org/api/querystring.html) так:

const querystring = require('querystring');

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

Вы также можете использовать библиотеку [qs](https://github.com/ljharb/qs).

**О версиях...**

До тех пор, пока Axios не достигнет версии «1.0», серьезные изменения автором данной библиотеки будут выпущены лишь с новой минорной версией. Например, 0.5.1 и 0.5.4 будут иметь один и тот же API, но версия 0.6.0 будет иметь в своем составе уже серьезные доработки.

**Promises / Промисы**

Работа axios напрямую зависит от поддержки Promise. Проверить подержку клиентами(браузерами) можно на сайте [can i use](http://caniuse.com/promises). Если Promise не поддерживаются, можно использовать [полифилл/polyfill](https://github.com/jakearchibald/es6-promise). Более подробно почитать о Promise на русском можно тут:

* [Learn JS](https://learn.javascript.ru/promise)
* [Руководство](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise)
* [Промисы на примерах из жизни](https://medium.com/web-standards/promises-explained-caee4c9b86d0)
* [Promise для новичков](https://habr.com/company/zerotech/blog/317256/)

**TypeScript**

Axios и [TypeScript](http://typescriptlang.org/) работают вместе

import axios from 'axios';

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