Для сетевых запросов из JavaScript есть широко известный термин «**AJAX**» (аббревиатура от Asynchronous JavaScript And XML). XML мы использовать не обязаны

**Fetch**

Метод **fetch()** — современный и очень мощный. Он не поддерживается старыми (можно использовать полифил), но поддерживается всеми современными браузерами. Базовый синтаксис:

let promise = fetch(url, [options])

* **url** – URL для отправки запроса.
* **options** – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url. Браузер сразу же начинает запрос и возвращает промис, который внешний код использует для получения результата.

Процесс получения ответа обычно происходит **в два этапа**:

**Во-первых**, promise выполняется с объектом встроенного класса [**Response**](https://fetch.spec.whatwg.org/#response-class) в качестве результата, как только сервер пришлёт **заголовки ответа**. На этом этапе мы можем проверить статус HTTP-запроса и определить, выполнился ли он успешно, а также посмотреть заголовки, но пока без тела ответа.

Промис завершается с ошибкой, если fetch не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 не являются ошибкой.

Мы можем увидеть HTTP-статус в **свойствах ответа:**

* status – код статуса HTTP-запроса, например 200.
* ok – логическое значение: будет true, если код HTTP-статуса в диапазоне 200-299.

Например:

let response = **await fetch**(url);

if (response.**ok**) { // если HTTP-статус в диапазоне 200-299

// получаем тело ответа (см. про этот метод ниже)

let json = await response.**json**();

} else {

alert("Ошибка HTTP: " + response.status);

}

**Во-вторых**, для получения **тела ответа** нам нужно использовать дополнительный вызов метода.

Response предоставляет несколько **методов**, основанных на промисах, для доступа к телу ответа в различных форматах:

* response.text() – читает ответ и возвращает как обычный текст,
* response.json() – декодирует ответ в формате JSON,
* response.formData() – возвращает ответ как объект FormData
* response.blob() – возвращает объект как Blob (бинарные данные с типом),
* response.arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),
* помимо этого, response.body – это объект **ReadableStream**, с помощью которого можно считывать тело запроса по частям.

Например, получим JSON-объект с последними коммитами из репозитория на GitHub:

let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits';

let response = **await** **fetch**(url);

let commits = **await** **response**.**json**(); // читаем ответ в формате JSON

alert(commits[0].author.login);

Пример работы с бинарными данными:

let response = await **fetch**('/article/fetch/logo-fetch.svg');

let blob = await response.**blob**(); // скачиваем как Blob-объект

// создаём <img>

let img = document.createElement('img');

img.style = 'position:fixed;top:10px;left:10px;width:100px';

document.body.append(img);

// выводим на экран

img.src = **URL.createObjectURL**(blob);

setTimeout(() => { // прячем через три секунды

img.remove();

URL.revokeObjectURL(img.src);

}, 3000);

**!** Мы можем выбрать **только один метод чтения ответа**.

**Итого**

Типичный запрос с помощью **fetch** состоит из двух операторов await:

let response = await fetch(url, options); // завершается с заголовками ответа

let result = await response.json(); // читать тело ответа в формате JSON

Или, без await:

fetch(url, options)

.then(response => response.json())

.then(result => /\* обрабатываем результат \*/)

### Заголовки ответа

**Заголовки ответа** хранятся в похожем на Map объекте **response.headers**. Это не совсем Map, но мы можем использовать такие же методы, как с Map, чтобы получить заголовок по его имени или перебрать заголовки в цикле:

let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');

// получить один заголовок

alert(response.**headers.get**('Content-Type')); // application/json; charset=utf-8

// перебрать все заголовки

for (let [key, value] of response.headers) {

alert(`${key} = ${value}`);

### Заголовки запроса

Для установки **заголовка запроса** в fetch мы можем использовать **опцию** **headers**. Она содержит объект с исходящими заголовками, например:

let response = fetch(protectedUrl, {

**headers**: {

Authentication: 'secret'

}

});

Есть список *запрещённых HTTP-заголовков*, которые мы не можем установить: Accept-Charset, Accept-Encoding, Access-Control-Request-Headers, Access-Control-Request-Method, Connection, Content-Length, Cookie, Cookie2, Date, DNT, Expect, Host, Keep-Alive, Origin, Referer, TE, Trailer, Transfer-Encoding, Upgrade, Via, Proxy-\*, Sec-\*

Эти заголовки обеспечивают достоверность данных и корректную работу протокола HTTP, поэтому они контролируются исключительно браузером.

### POST-запросы

Для отправки POST-запроса или запроса с другим методом, нам необходимо использовать **fetch параметры**:

method – HTTP метод, например POST,

body – тело запроса, одно из списка:

* **строка** (например, в формате JSON),
* объект **FormData** для отправки данных как form/multipart,
* **Blob/BufferSource** для отправки бинарных данных,
* **URLSearchParams** для отправки данных в кодировке x-www-form-urlencoded, используется редко.

Чаще всего используется JSON. Например, этот код отправляет объект user как JSON:

let user = {

name: 'John',

surname: 'Smith'

};

let response = await fetch('/article/fetch/post/user', {

**method**: 'POST',

**headers**: {

'Content-Type': 'application/json;charset=utf-8'

},

**body**: JSON.stringify(user)

});

let result = await **response.json**();

alert(result.message);

Заметим, что так как тело запроса body – строка, то заголовок Content-Type по умолчанию будет **text/plain;charset=UTF-8**. Но, так как мы посылаем JSON, то используем параметр headers для отправки вместо этого **application/json**, правильный Content-Type для JSON.

### Fetch: ход загрузки

Метод fetch позволяет отслеживать процесс *получения* данных.

Заметим, на данный момент в fetch нет способа отслеживать процесс *отправки*. Для этого используйте [XMLHttpRequest](https://learn.javascript.ru/xmlhttprequest).

Чтобы отслеживать ход загрузки данных с сервера, можно использовать свойство response.body. Это **ReadableStream** («поток для чтения») – особый объект, который предоставляет тело ответа по частям, по мере поступления. Потоки для чтения описаны в спецификации [Streams API](https://streams.spec.whatwg.org/#rs-class).

В отличие от response.text(), response.json() и других методов, response.body даёт полный контроль над процессом чтения, и мы можем подсчитать, сколько данных получено на каждый момент. Вот примерный код, который читает ответ из response.body:

// вместо response.json() и других методов

const reader **= response.body.getReader()**;

// бесконечный цикл, пока идёт загрузка

**while(true)** {

// done становится true в последнем фрагменте

// value - Uint8Array из байтов каждого фрагмента

const {done, value} = await **reader.read();**

if (done) {

break;

}

console.log(`Получено ${value.length} байт`)

}

Результат вызова await **reader.read()** – это объект с двумя свойствами:

* **done** – true, когда чтение закончено, иначе false.
* **value** – типизированный массив данных ответа **Uint8Array**

### Fetch: прерывание запроса

Как мы знаем, метод fetch возвращает промис. А в JavaScript в целом нет понятия «отмены» промиса. Как же прервать запрос fetch? Для таких целей существует специальный встроенный объект**: AbortController**, который можно использовать для отмены не только fetch, но и других асинхронных задач.

Использовать его достаточно просто:

**Шаг 1**: создаём контроллер:

let controller = **new AbortController();**

Контроллер имеет единственный метод abort() и единственное свойство signal. При вызове abort():

* генерируется событие с именем abort на объекте **controller.signal**
* свойство **controller.signal.aborted** становится равным **true**.

Все, кто хочет узнать о вызове abort(), ставят обработчики на controller.signal, чтобы отслеживать его.

let controller = **new AbortController**();

let signal = **controller.signal**;

// срабатывает при вызове controller.abort()

signal.addEventListener(**'abort'**, () => alert("отмена!"));

**controller**.**abort**(); // отмена!

alert(**signal**.**aborted**); // true

**Шаг 2**: передайте **свойство signal** опцией в метод **fetch**:

let controller = new AbortController();

fetch(url, {

**signal**: controller.signal

});

Метод fetch умеет работать с AbortController, он **слушает событие abort на signal**.

**Шаг 3**: чтобы прервать выполнение fetch, вызовите **controller.abort():**

controller.abort();

Когда fetch отменяется, его промис завершается с ошибкой **AbortError**, поэтому мы должны обработать её, например, в try..catch

AbortController – **масштабируемый**, он позволяет отменить несколько вызовов fetch одновременно. Если у нас есть собственные асинхронные задачи, отличные от fetch, мы можем использовать один AbortController для их остановки вместе с fetch. Нужно лишь слушать его событие abort

let urls = [...];

let controller = new AbortController();

let ourJob = new Promise((resolve, reject) => { // наша задача

...

controller.signal.addEventListener('abort', reject);

});

let fetchJobs = urls.map(url => fetch(url, { // запросы fetch

signal: controller.signal

}));

// ожидать выполнения нашей задачи и всех запросов

let results = await Promise.all([...fetchJobs, ourJob]);

// вызов откуда-нибудь ещё:

// controller.abort() прервёт все вызовы fetch и наши задачи