

Раздел

Сетевые запросы

Навигация по уроку

Зачем нужен CORS? Экскурс в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого


Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub

 → Сетевые запросы 2-го октября 2020

Fetch: запросы на другие сайты

Если мы сделаем запрос `fetch` на другой веб-сайт, он, вероятно, завершится неудачей.

Например, давайте попробуем запросить `http://example.com`:

```
1 try {
2   await fetch('http://example.com');
3 } catch(err) {
4   alert(err); // Failed to fetch
5 }
```



Вызов `fetch` не удался, как и ожидалось.

Ключевым понятием здесь является *источник* (*origin*) – комбинация домен/порт/протокол.

Запросы на другой источник – отправленные на другой домен (или даже поддомен), или протокол, или порт – требуют специальных заголовков от удалённой стороны.

Эта политика называется «CORS»: Cross-Origin Resource Sharing («совместное использование ресурсов между разными источниками»).

Зачем нужен CORS? Экскурс в историю

CORS существует для защиты интернета от злых хакеров.

Серьёзно. Давайте сделаем краткое историческое отступление.

Многие годы скрипт с одного сайта не мог получить доступ к содержимому другого сайта.

Это простое, но могучее правило было основой интернет-безопасности. Например, хакерский скрипт с сайта `hacker.com` не мог получить доступ к почтовому ящику пользователя на сайте `gmail.com`. И люди чувствовали себя спокойно.

В то время в JavaScript не было методов для сетевых запросов. Это был «игрушечный» язык для украшения веб-страниц.

Но веб-разработчики жаждали большей власти. Чтобы обойти этот запрет и всё же получать данные с других сайтов, были придуманы разные хитрости.

Использование форм

Одним из способов общения с другим сервером была отправка туда формы `<form>`. Люди отправляли её в `<iframe>`, чтобы оставаться на текущей странице, вот так:

```
1 <!-- цель формы -->
2 <iframe name="iframe"></iframe>
3
4 <!-- форма могла быть динамически сгенерирована и отпра
5 <form target="iframe" method="POST" action="http://anot
6   ...
7 </form>
```

Таким способом было возможно сделать GET/POST запрос к другому сайту даже без сетевых методов, так как формы можно отправлять куда угодно. Но так как запрещено получать доступ к содержимому `<iframe>` с другого сайта, прочитать ответ было невозможно.

Раздел

Сетевые запросы

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Если быть точным, были трюки и для этого, требующие специального кода на странице и в ифрейме, так что общение с ифреймом было технически возможно. Сейчас мы не будем вдаваться в подробности, пусть эти динозавры покоятся в мире.

Использование скриптов

Ещё один трюк заключался в использовании тега `script`. У него может быть любой `src`, с любым доменом, например `<script src="http://another.com/...">`. Это даёт возможность загрузить и выполнить скрипт откуда угодно.

Если сайт, например `another.com`, хотел предоставить данные для такого доступа, он предоставлял так называемый «протокол JSONP» (**JSON with Padding**).

Вот как он работал.

Например, нам на нашем сайте нужны данные с сайта `http://another.com`, скажем, погода:

1. Сначала, заранее, объявляем глобальную функцию для обработки данных, например `gotWeather`.

```
1 // 1. Объявить функцию для обработки погодных данных
2 function gotWeather({ temperature, humidity }) {
3   alert(`температура: ${temperature}, влажность: ${hu
4 }
```

2. Затем создаём тег `<script>` с `src="http://another.com/weather.json?callback=gotWeather"`, при этом имя нашей функции – в URL-парамetre `callback`.

```
1 let script = document.createElement('script');
2 script.src = `http://another.com/weather.json?callbac
3 document.body.append(script);
```

3. Удалённый сервер с `another.com` должен в ответ сгенерировать скрипт, который вызывает `gotWeather(...)` с данными, которые хочет передать.

```
1 // Ожидаемый ответ от сервера выглядит так:
2 gotWeather({
3   temperature: 25,
4   humidity: 78
5 });
```

4. Когда этот скрипт загрузится и выполнится, наша функция `gotWeather` получает данные.

Это работает и не нарушает безопасность, потому что обе стороны согласились передавать данные таким образом. А когда обе стороны согласны, то это определённо не хак. Всё ещё существуют сервисы, которые предоставляют такой доступ, так как это работает даже для очень старых браузеров.

Спустя некоторое время в браузерном JavaScript появились методы для сетевых запросов.

Вначале запросы на другой источник были запрещены. Но в результате долгих дискуссий было решено разрешить их делать, но для использования новых возможностей требовать разрешение сервера, выраженное в специальных заголовках.

Простые запросы

Есть два вида запросов на другой источник:

1. Простые.

Раздел

Сетевые запросы

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



2. Все остальные.

Простые запросы будут попроще, поэтому давайте начнём с них.

Простой запрос – это запрос, удовлетворяющий следующим условиям:

1. **Простой метод**: GET, POST или HEAD
2. **Простые заголовки** – разрешены только:
 - Accept ,
 - Accept-Language ,
 - Content-Language ,
 - Content-Type со значением application/x-www-form-urlencoded, multipart/form-data или text/plain .

Любой другой запрос считается «непростым». Например, запрос с методом PUT или с HTTP-заголовком API-Key не соответствует условиям.

Принципиальное отличие между ними состоит в том, что «простой запрос» может быть сделан через <form> или <script>, без каких-то специальных методов.

Таким образом, даже очень старый сервер должен быть способен принять простой запрос.

В противоположность этому, запросы с нестандартными заголовками или, например, методом DELETE нельзя создать таким способом. Долгое время JavaScript не мог делать такие запросы. Поэтому старый сервер может предположить, что такие запросы поступают от привилегированного источника, «просто потому, что веб-страница неспособна их посылать».

Когда мы пытаемся сделать непростой запрос, браузер посылает специальный предварительный запрос («предзапрос», по англ. «preflight»), который спрашивает у сервера – согласен ли он принять такой непростой запрос или нет?

И, если сервер явно не даёт согласие в заголовках, непростой запрос не посылается.

Далее мы разберём конкретные детали.

CORS для простых запросов

При запросе на другой источник браузер всегда ставит «от себя» заголовок Origin .

Например, если мы запрашиваем `https://anywhere.com/request` со страницы `https://javascript.info/page`, заголовки будут такими:

```
1 GET /request
2 Host: anywhere.com
3 Origin: https://javascript.info
4 ...
```

Как вы можете видеть, заголовок Origin содержит именно источник (домен/протокол/порт), без пути.

Сервер может проверить Origin и, если он согласен принять такой запрос, добавить особый заголовок Access-Control-Allow-Origin к ответу. Этот заголовок должен содержать разрешённый источник (в нашем случае `https://javascript.info`) или звёздочку *. Тогда ответ успешен, в противном случае возникает ошибка.

Здесь браузер играет роль доверенного посредника:

1. Он гарантирует, что к запросу на другой источник добавляется правильный заголовок Origin .
2. Он проверяет наличие разрешающего заголовка Access-Control-Allow-Origin в ответе и, если всё хорошо, то JavaScript получает доступ к ответу сервера, в противном случае – доступ запрещается с ошибкой.

JavaScript

Браузер

Сервер

Раздел

Сетевые запросы

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

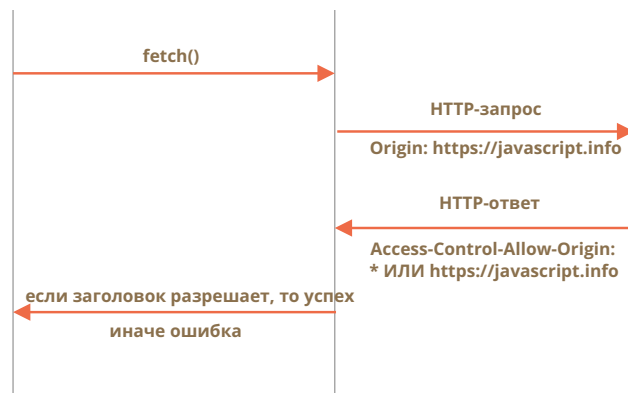
Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Вот пример ответа сервера, который разрешает доступ:

```
1 200 OK
2 Content-Type:text/html; charset=UTF-8
3 Access-Control-Allow-Origin: https://javascript.info
```

Заголовки ответа

По умолчанию при запросе к другому источнику JavaScript может получить доступ только к так называемым «простым» заголовкам ответа:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

При доступе к любому другому заголовку ответа будет ошибка.

i Обратите внимание: нет Content-Length

Пожалуйста, обратите внимание: в списке нет заголовка Content-Length!

Этот заголовок содержит полную длину ответа. Поэтому если мы загружаем что-то и хотели бы отслеживать прогресс в процентах, то требуется дополнительное разрешение для доступа к этому заголовку (читайте ниже).

Чтобы разрешить JavaScript доступ к любому другому заголовку ответа, сервер должен указать заголовок Access-Control-Expose-Headers. Он содержит список, через запятую, заголовков, которые не являются простыми, но доступ к которым разрешён.

Например:

```
1 200 OK
2 Content-Type:text/html; charset=UTF-8
3 Content-Length: 12345
4 API-Key: 2c9de507f2c54aa1
5 Access-Control-Allow-Origin: https://javascript.info
6 Access-Control-Expose-Headers: Content-Length,API-Key
```

При таком заголовке Access-Control-Expose-Headers, скрипту разрешено получить заголовки Content-Length и API-Key ответа.

«Непростые» запросы

Мы можем использовать любой HTTP-метод: не только GET/POST, но и PATCH, DELETE и другие.

Раздел

[Сетевые запросы](#)

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Некоторое время назад никто не мог даже предположить, что веб-страница способна делать такие запросы. Так что могут существовать веб-сервисы, которые рассматривают нестандартный метод как сигнал: «Это не браузер». Они могут учитывать это при проверке прав доступа.

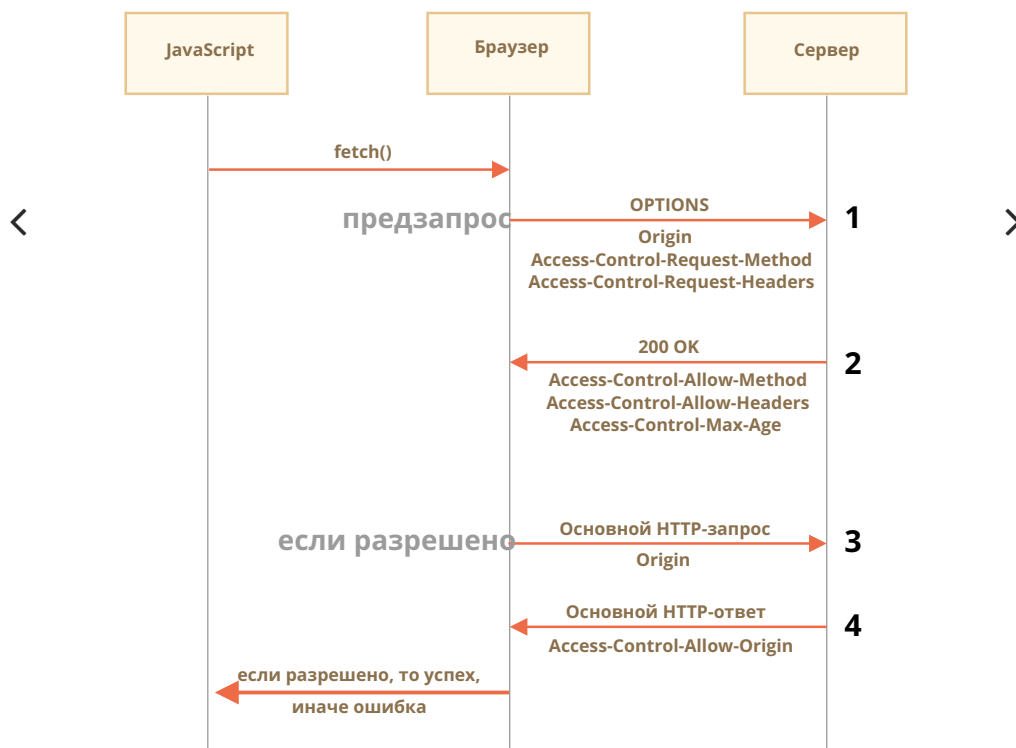
Поэтому, чтобы избежать недопониманий, браузер не делает «непростые» запросы (которые нельзя было сделать в прошлом) сразу. Перед этим он посылает предварительный запрос, спрашивая разрешения.

Предварительный запрос использует метод `OPTIONS`, у него нет тела, но есть два заголовка:

- `Access-Control-Request-Method` содержит HTTP-метод «непростого» запроса.
- `Access-Control-Request-Headers` предоставляет разделённый запятыми список его «непростых» HTTP-заголовков.

Если сервер согласен принимать такие запросы, то он должен ответить без тела, со статусом 200 и с заголовками:

- `Access-Control-Allow-Methods` должен содержать разрешённые методы.
- `Access-Control-Allow-Headers` должен содержать список разрешённых заголовков.
- Кроме того, заголовок `Access-Control-Max-Age` может указывать количество секунд, на которое нужно кешировать разрешения. Так что браузеру не придётся посылать предзапрос для последующих запросов, удовлетворяющих данным разрешениям.



Давайте пошагово посмотрим, как это работает, на примере `PATCH` запроса (этот метод часто используется для обновления данных) на другой источник:

```
1 let response = await fetch('https://site.com/service.js',
2   method: 'PATCH',
3   headers: {
4     'Content-Type': 'application/json'
5     'API-Key': 'secret'
6   }
7 });
```

Этот запрос не является простым по трём причинам (достаточно одной):

Раздел

Сетевые запросы

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



- Метод PATCH
- Content-Type не один из: application/x-www-form-urlencoded, multipart/form-data, text/plain.
- Содержит «непростой» заголовок API-Key.

Шаг 1 (предзапрос)

Перед тем, как послать такой запрос, браузер самостоятельно генерирует и посылает предзапрос, который выглядит следующим образом:

```
1 OPTIONS /service.json
2 Host: site.com
3 Origin: https://javascript.info
4 Access-Control-Request-Method: PATCH
5 Access-Control-Request-Headers: Content-Type, API-Key
```

- Метод: OPTIONS.
- Путь – точно такой же, как в основном запросе: /service.json.
- Особые заголовки:
 - Origin – источник.
 - Access-Control-Request-Method – запрашиваемый метод.
 - Access-Control-Request-Headers – разделённый запятыми список «непростых» заголовков запроса.

Шаг 2 (ответ сервера на предзапрос)

Сервер должен ответить со статусом 200 и заголовками:

- Access-Control-Allow-Methods: PATCH
- Access-Control-Allow-Headers: Content-Type, API-Key.



Это разрешит будущую коммуникацию, в противном случае возникает ошибка.



Если сервер ожидает в будущем другие методы и заголовки, то он может в ответе перечислить их все сразу, разрешить заранее, например:

```
1 200 OK
2 Access-Control-Allow-Methods: PUT, PATCH, DELETE
3 Access-Control-Allow-Headers: API-Key, Content-Type, If-M
4 Access-Control-Max-Age: 86400
```

Теперь, когда браузер видит, что PATCH есть в Access-Control-Allow-Methods, а Content-Type, API-Key – в списке Access-Control-Allow-Headers, он посылает наш основной запрос.

Кроме того, ответ на предзапрос кешируется на время, указанное в заголовке Access-Control-Max-Age (86400 секунд, один день), так что последующие запросы не вызовут предзапрос. Они будут отосланы сразу при условии, что соответствуют закешированным разрешениям.

Шаг 3 (основной запрос)

Если предзапрос успешен, браузер делает основной запрос. Алгоритм здесь такой же, что и для простых запросов.

Основной запрос имеет заголовок Origin (потому что он идёт на другой источник):

```
1 PATCH /service.json
2 Host: site.com
3 Content-Type: application/json
4 API-Key: secret
5 Origin: https://javascript.info
```

Раздел

Сетевые запросы

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub

Шаг 4 (основной ответ)

Сервер не должен забывать о добавлении `Access-Control-Allow-Origin` к ответу на основной запрос. Успешный предзапрос не освобождает от этого:

```
1 Access-Control-Allow-Origin: https://javascript.info
```

После этого JavaScript может прочитать ответ сервера.

На заметку:

Предзапрос осуществляется «за кулисами», невидимо для JavaScript.

JavaScript получает только ответ на основной запрос или ошибку, если со стороны сервера нет разрешения.

Авторизационные данные

Запрос на другой источник по умолчанию не содержит авторизационных данных (`credentials`), под которыми здесь понимаются куки и заголовки HTTP-аутентификации.

Это нетипично для HTTP-запросов. Обычно запрос к `http://site.com` сопровождается всеми куки с этого домена. Но запросы на другой источник, сделанные методами JavaScript – исключение.

Например, `fetch('http://another.com')` не посылает никаких куки, даже тех (!), которые принадлежат домену `another.com`.

Почему?

Потому что запрос с авторизационными данными даёт намного больше возможностей, чем без них. Если он разрешён, то это позволяет JavaScript действовать от имени пользователя и получать информацию, используя его авторизационные данные.

Действительно ли сервер настолько доверяет скрипту? Тогда он должен явно разрешить такие запросы при помощи дополнительного заголовка.

Чтобы включить отправку авторизационных данных в `fetch`, нам нужно добавить опцию `credentials: "include"`, вот так:

```
1 fetch('http://another.com', {
2   credentials: "include"
3 });
```

Теперь `fetch` пошлёт куки с домена `another.com` вместе с нашим запросом на этот сайт.

Если сервер согласен принять запрос с авторизационными данными, он должен добавить заголовок `Access-Control-Allow-Credentials: true` к ответу, в дополнение к `Access-Control-Allow-Origin`.

Например:

```
1 200 OK
2 Access-Control-Allow-Origin: https://javascript.info
3 Access-Control-Allow-Credentials: true
```

Пожалуйста, обратите внимание: в `Access-Control-Allow-Origin` запрещено использовать звёздочку `*` для запросов с авторизационными данными. Там должен быть именно источник, как показано выше. Это дополнительная мера безопасности, чтобы гарантировать, что сервер действительно знает, кому он доверяет делать такие запросы.

Итого

Раздел

Сетевые запросы

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



С точки зрения браузера запросы к другому источнику бывают двух видов: «простые» и все остальные.

Простые запросы должны удовлетворять следующим условиям:

- Метод: GET, POST или HEAD.
- Заголовки – мы можем установить только:
 - Accept
 - Accept-Language
 - Content-Language
 - Content-Type со значением `application/x-www-form-urlencoded`, `multipart/form-data` или `text/plain`.

Основное их отличие заключается в том, что простые запросы с давних времён выполнялись с использованием тегов `<form>` или `<script>`, в то время как непростые долгое время были невозможны для браузеров.

Практическая разница состоит в том, что простые запросы отправляются сразу с заголовком `Origin`, а для других браузер делает предварительный запрос, спрашивая разрешения.

Для простых запросов:

- → Браузер посылает заголовок `Origin` с источником.
- ← Для запросов без авторизационных данных (не отправляются умолчанию) сервер должен установить:
 - `Access-Control-Allow-Origin` в `*` или то же значение, что и `Origin`
- ← Для запросов с авторизационными данными сервер должен установить:
 - `Access-Control-Allow-Origin` в то же значение, что и `Origin`
 - `Access-Control-Allow-Credentials` в `true`

Дополнительно, чтобы разрешить JavaScript доступ к любым заголовкам ответа, кроме `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified` или `Pragma`, сервер должен перечислить разрешённые в заголовке `Access-Control-Expose-Headers`.

Для непростых запросов перед основным запросом отправляется предзапрос:

- → Браузер посылает запрос `OPTIONS` на тот же адрес с заголовками:
 - `Access-Control-Request-Method` – содержит запрашиваемый метод,
 - `Access-Control-Request-Headers` – перечисляет непростые запрашиваемые заголовки.
- ← Сервер должен ответить со статусом 200 и заголовками:
 - `Access-Control-Allow-Methods` со списком разрешённых методов,
 - `Access-Control-Allow-Headers` со списком разрешённых заголовков,
 - `Access-Control-Max-Age` с количеством секунд для кэширования разрешений
- → Затем отправляется основной запрос, применяется предыдущая «простая» схема.

✓ Задачи

Почему нам нужен Origin?

важность: 5

Как вы, вероятно, знаете, существует HTTP-заголовок `Referer`, который обычно содержит адрес страницы, инициировавшей сетевой запрос.

Например, при запросе (fetch) `http://google.com` с `http://javascript.info/some/url` заголовки выглядят так:

1 Accept: */*

Раздел

[Сетевые запросы](#)

Навигация по уроку

Зачем нужен CORS? Экскурсия в историю

Простые запросы

CORS для простых запросов

Заголовки ответа

«Непростые» запросы

Авторизационные данные

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
2 Accept-Charset: utf-8
3 Accept-Encoding: gzip,deflate,sdch
4 Connection: keep-alive
5 Host: google.com
6 Origin: http://javascript.info
7 Referer: http://javascript.info/some/url
```

Как вы можете видеть, присутствуют и `Referer`, и `Origin`.

Вопросы:

1. Почему нужен `Origin`, если `Referer` содержит даже больше информации?
2. Возможно ли отсутствие `Referer` или `Origin`, или это неправильно?

решение

Проводим [курсы по JavaScript и фреймворкам](#).

Комментарии

перед тем как писать...

