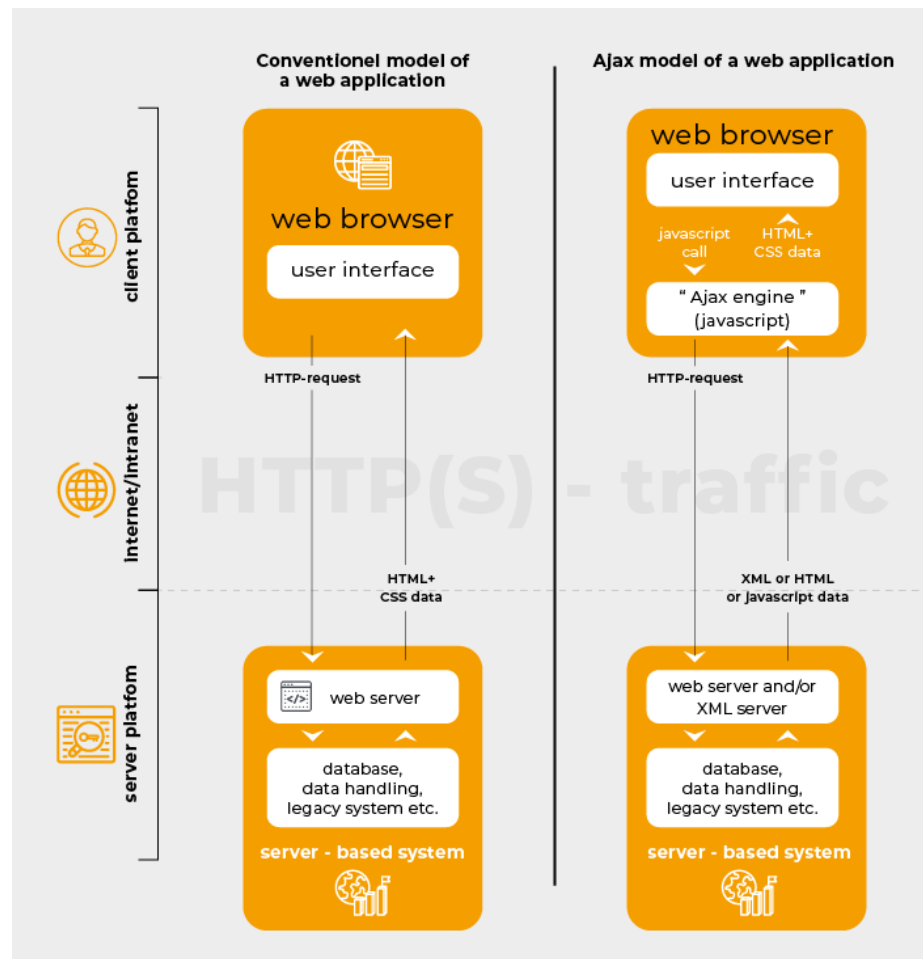
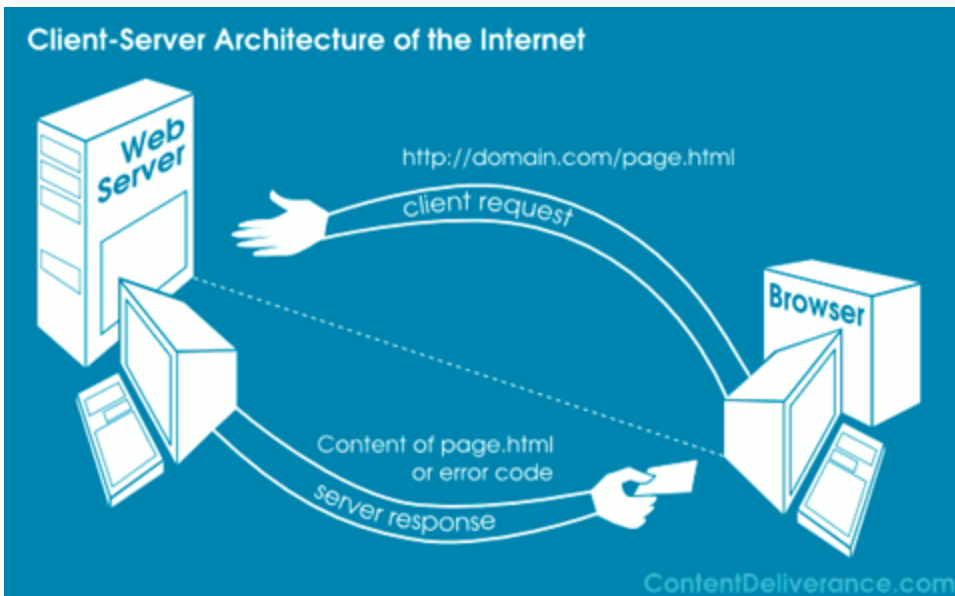


Лекция 6

AJAX

Интернет, HTTP и AJAX



Подробнее тут <https://developer.mozilla.org/ru/docs/Web/Guide/AJAX>

Аjax

Аjax означает Асинхронный JavaScript и XML.

В основе технологии лежит использование XMLHttpRequest, который позволяет нам отправлять и получать информацию в различных форматах включая XML, HTML и даже текстовые файлы.

Самое привлекательное в Ajax — это его асинхронный принцип работы.

С помощью этой технологии можно осуществлять взаимодействие с сервером без необходимости перезагрузки страницы. Это позволяет обновлять содержимое страницы частично, в зависимости от действий пользователя.



Аякс Великий
герой Троянской войны

Что передаем по сети

- Текстовые данные
- JSON
- Бинарные данные
- Файлы

JSON - это общий формат для представления значений и объектов.

Первоначально он был создан для JavaScript, но многие другие языки также имеют библиотеки, которые могут работать с ним.

```
1 {  
2   "name": "John",  
3   "age": 30,  
4   "isAdmin": false,  
5   "courses": ["html", "css", "js"],  
6   "wife": null  
7 }
```

Blob и FormData

Blob - это объект в JavaScript, который позволяет работать с бинарными данными.

```
1 const binary = new Uint8Array(1024 * 1024); // 1 MB данных
2 const blob = new Blob([binary], {type: 'application/octet-stream'});
```

FormData - объект JavaScript, который позволяет работать с формами и файлами.

```
1 const fileInput = document.querySelector('input[type=file]');
2 const file = fileInput.files[0].file;
3 const formdata = new FormData();
4 formdata.append('file', file);
```

XMLHttpRequest

XMLHttpRequest (XHR) – это встроенный в браузер объект, который даёт возможность делать HTTP-запросы к серверу без перезагрузки страницы.

XHR поддерживает 2 режима работы:

- Синхронный
- Асинхронный

XMLHttpRequest

- Создаем XHR

```
1 let xhr = new XMLHttpRequest();
```

- Инициализируем его

```
1 xhr.open('GET', '/article/xmlhttprequest/example/load');
```

- Посылаем запрос

```
1 xhr.send();
```

- Слушаем ответ

XMLHttpRequest (слушаем ответ)

После получения ответа
мы можем достать его из
xhr

- **status** - HTTP статус ответа
- **statusText** - текстовых HTTP статус ответа
- **response** - тело ответа

```
1 xhr.onload = function() {  
2   alert(`Загружено: ${xhr.status} ${xhr.response}`);  
3 };  
4  
5 xhr.onerror = function() {  
6   alert(`Ошибка соединения`);  
7 };  
8  
9 xhr.onprogress = function(event) {  
10  alert(`Загружено ${event.loaded} из ${event.total}`);  
11 };
```


XMLHttpRequest (асинхронный запрос)

```
1 // 1. Создаём новый XMLHttpRequest-объект
2 let xhr = new XMLHttpRequest();
3
4 // 2. Настраиваем его: GET-запрос по URL /article/.../load
5 xhr.open('GET', '/article/xmlhttprequest/example/load');
6
7 // 3. Отсылаем запрос
8 xhr.send();
9
10 // 4. Этот код сработает после того, как мы получим ответ сервера
11 xhr.onload = function() {
12     if (xhr.status !== 200) { // анализируем HTTP-статус ответа, если статус не 200, то произошла ошибка
13         alert(`Ошибка ${xhr.status}: ${xhr.statusText}`); // Например, 404: Not Found
14     } else { // если всё прошло гладко, выводим результат
15         alert(`Готово, получили ${xhr.response.length} байт`); // response -- это ответ сервера
16     }
17 };
18
19 xhr.onerror = function() {
20     alert("Запрос не удался");
21 };
```

XMLHttpRequest (синхронный запрос)

```
1 let xhr = new XMLHttpRequest();
2
3 xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);
4
5 try {
6   xhr.send();
7   if (xhr.status != 200) {
8     alert(`Ошибка ${xhr.status}: ${xhr.statusText}`);
9   } else {
10    alert(xhr.response);
11  }
12 } catch(err) { // для отлова ошибок используем конструкцию try...catch вместо onerror
13   alert("Запрос не удался");
14 }
```

XMLHttpRequest (тип ответа)

- text - строка
- arrayBuffer - ArrayBuffer бинарные данные
- blob - Blob бинарные данные
- document - XML-документ
- json - JSON (автоматически парситься)

```
1 xhr.responseType = 'json';
```

XMLHttpRequest (состояние запроса)

- UNSET = 0 - исходное состояние
- OPENED = 1 - вызван метод open
- HEADERS_RECEIVED = 2 - получены заголовки ответа
- LOADING = 3 - ответ в процессе передачи
- DONE = 4 - запрос завершен

```
get(url, callback) {  
    const xhr = new XMLHttpRequest();  
    xhr.open('GET', url);  
    xhr.send();  
  
    xhr.onreadystatechange = () => {  
        if (xhr.readyState === 4) {  
            this._handleResponse(xhr, callback);  
        }  
    };  
}
```

XMLHttpRequest (POST-запрос)

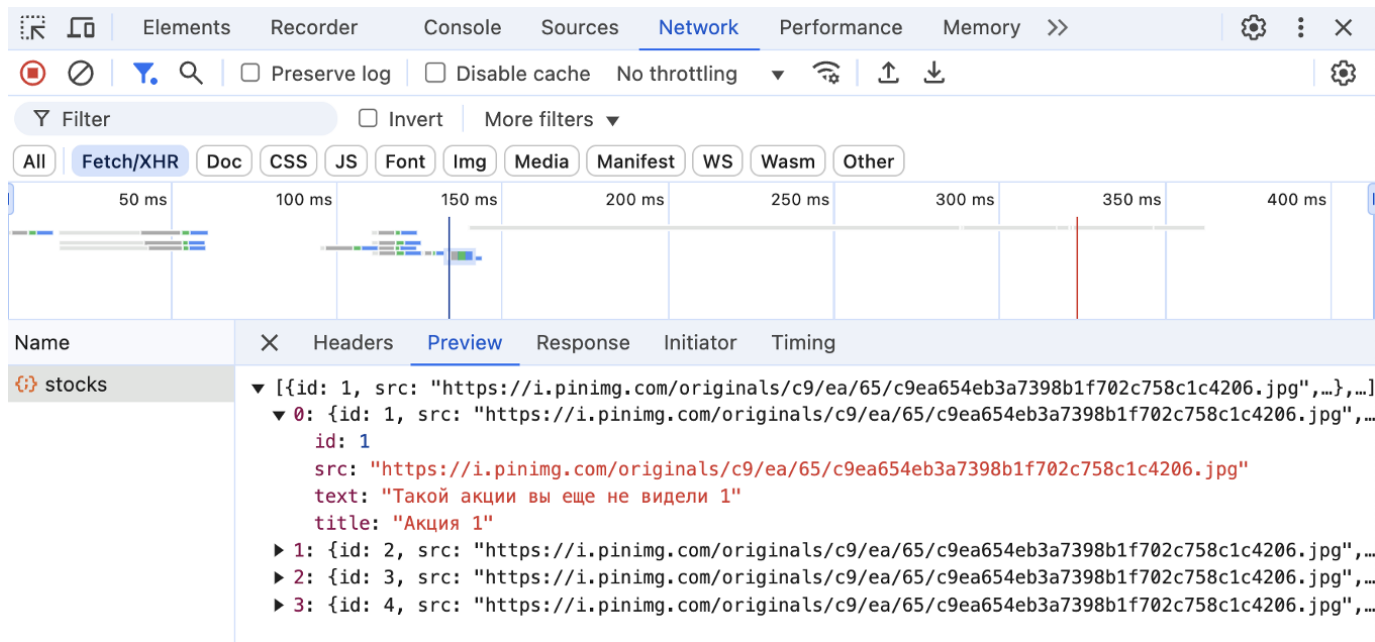
```
* @param {string} url - Адрес запроса
* @param {object} data - Данные для отправки
* @param {function} callback - Функция обратного вызова (data, status)
*/
post(url, data, callback) {
  const xhr = new XMLHttpRequest();
  xhr.open('POST', url);
  xhr.setRequestHeader('Content-Type', 'application/json');
  xhr.send(JSON.stringify(data));

  xhr.onreadystatechange = () => {
    if (xhr.readyState === 4) {
      this._handleResponse(xhr, callback);
    }
  };
}
```

```
// POST пример
api.post('https://api.example.com/create', { name: 'John' }, (data, status) => {
  console.log(status, data);
});
```

Вкладка Network

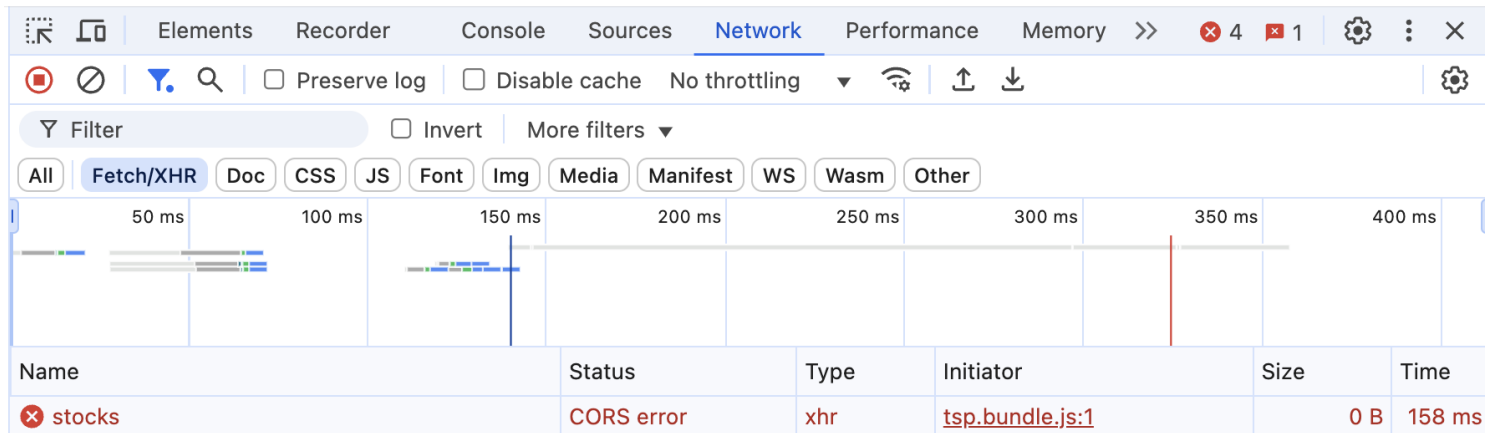
- Оставляем фильтр Fetch/XHR, о Fetch мы поговорим на другой лекции
- Но чтобы получить этот результат нам нужно еще понять **Same Origin Policy**



The screenshot shows the Chrome DevTools Network tab. The 'Fetch/XHR' filter is selected. The 'stocks' request is expanded, showing a JSON response. The response is an array of objects, each representing a stock. The first object is expanded, showing its properties: id, src, text, and title.

```
▼ [{"id": 1, "src": "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg", "..."}, ...]  
  ▼ 0: {"id": 1, "src": "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg", ...  
      id: 1  
      src: "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg"  
      text: "Такой акции вы еще не видели 1"  
      title: "Акция 1"  
    }  
    ► 1: {"id": 2, "src": "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg", ...  
    ► 2: {"id": 3, "src": "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg", ...  
    ► 3: {"id": 4, "src": "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg", ...
```

CORS



sts 0 B / 1.6 MB transferred 0 B / 1.9 MB resources Finish: 354 ms DOMContentLoaded: 136 ms Load: 321 ms

top Filter

Default levels

1 Issue: 1

Messages

er messages

ors

warnings

Attempting initialization Sat Apr 12 2025 19:23:52 GMT+0300 (Москва, стандартное время)

[content-script.ts:82](#)

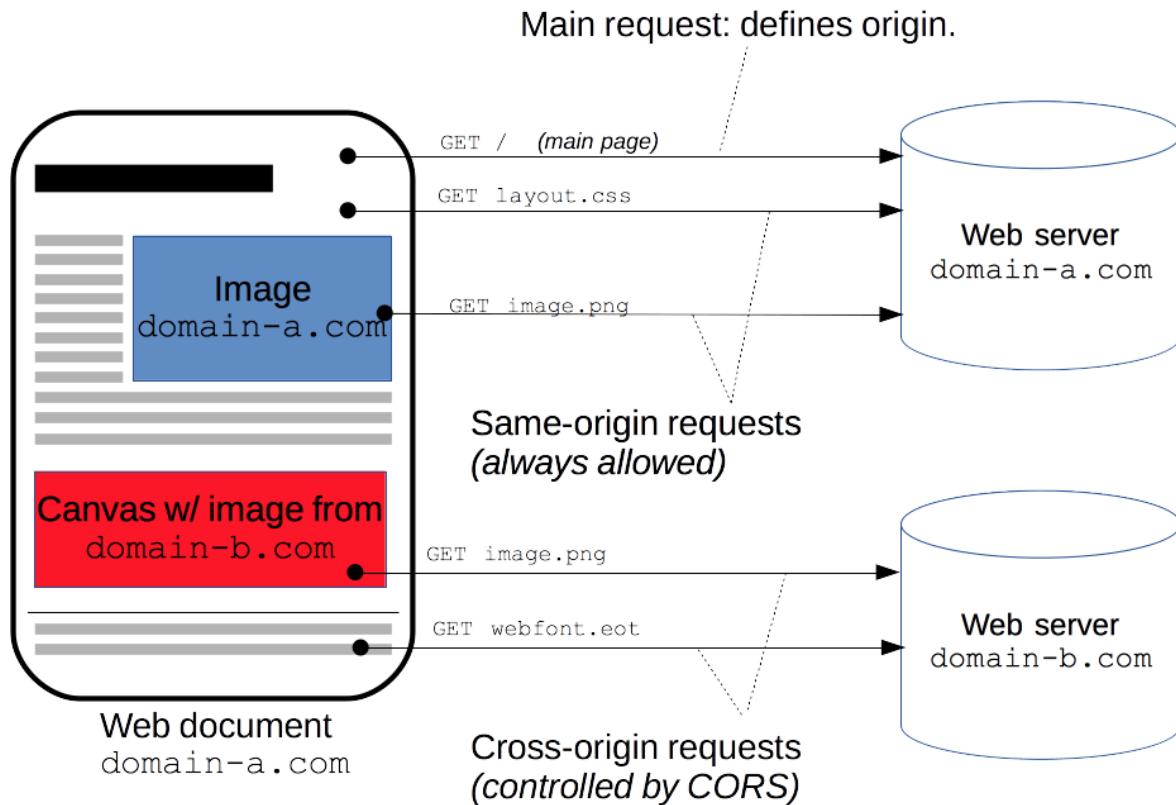
✖ Access to XMLHttpRequest at '<http://localhost:3000/stocks>' from origin '<http://127.0.0.1:5501>' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

CORS

- CORS - мы получили страницу с одного домена, а запросы отправляем на другой

Как решить?

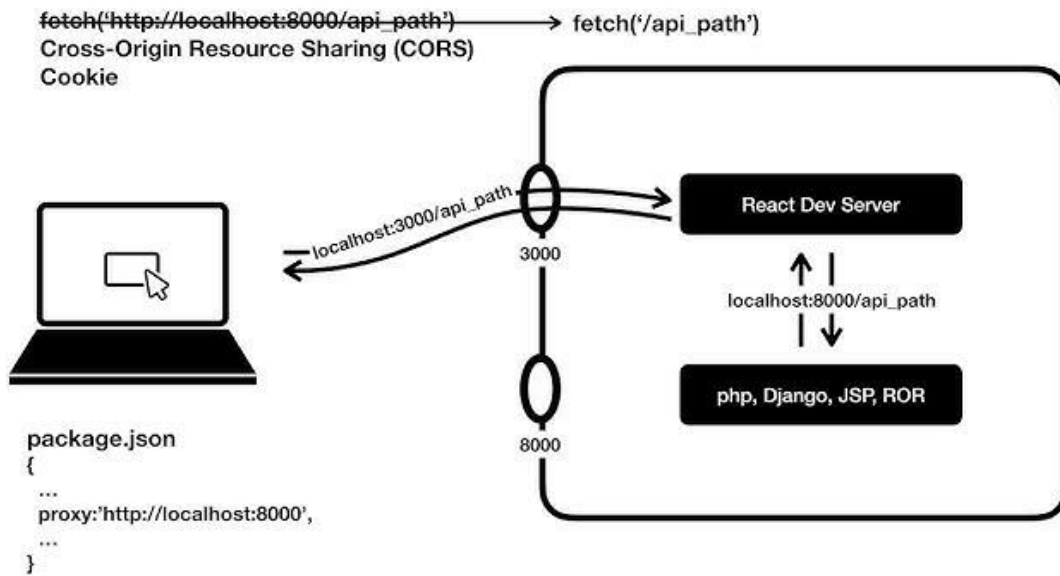
- CORS – заголовки на бекенде
- Проксирование через сервер фронтенда



Подробнее тут <https://learn.javascript.ru/fetch-crossorigin>

Обратный прокси-сервер для CORS

- Одно из решений - отправляем запросы напрямую в веб-сервис, а проксируем через наш сервер фронтенда
- Похоже на prod решение при проксировании через Nginx
- Будем применять на курсе РИП



Same Origin Policy

Same Origin Policy (правило ограничения домена) - это важная концепция безопасности работы web-приложения. Она призвана ограничивать возможности пользовательских сценариев из определенного источника по доступу к ресурсам и информации из других источников.

В нашем курсе мы будем обходить данное ограничение самым простым способом

- в ЛР-5 с помощью расширения браузера мы просто игнорируем эти правила
- в ЛР-6 мы сделаем один и тот же домен у бэкенда и фронтенда (bundle)

Same Origin Policy

Два URL считаются имеющими **один источник** (“same origin”), если у них одинаковый протокол, домен и порт

- `http://site.com`
- `http://site.com/`
- `http://site.com/my/page.html`

Разный источник:

- `http://site.com`
- `http://www.site.com` (другой домен)
- `http://site.org` (другой домен)
- `https://site.com` (другой протокол)
- `http://site.com:8080` (другой порт)

Same Origin Policy (типы взаимодействия с ресурсами)

- Запись в ресурсы - переходы по ссылкам, редиректы, сабмит форм
 - Встраивание ресурсов в другие ресурсы - добавление JavaScript, CSS, Images и тд с помощью HTML тегов
 - Чтение из других ресурсов - чтение ответов на запросы, получение доступа к содержимому встроенного ресурса
-
- Cross Origin “Запись” - обычно разрешена
 - Cross Origin “Встраивание” - обычно разрешено
 - Cross Origin “Чтение” - по-умолчанию запрещено

Cross-Origin Resource Sharing (**CORS**) standard — спецификация, позволяющая обойти ограничения, которые Same Origin Policy накладывает на кросс-доменные запросы

CORS (браузер)

Браузер играет роль доверенного посредника:

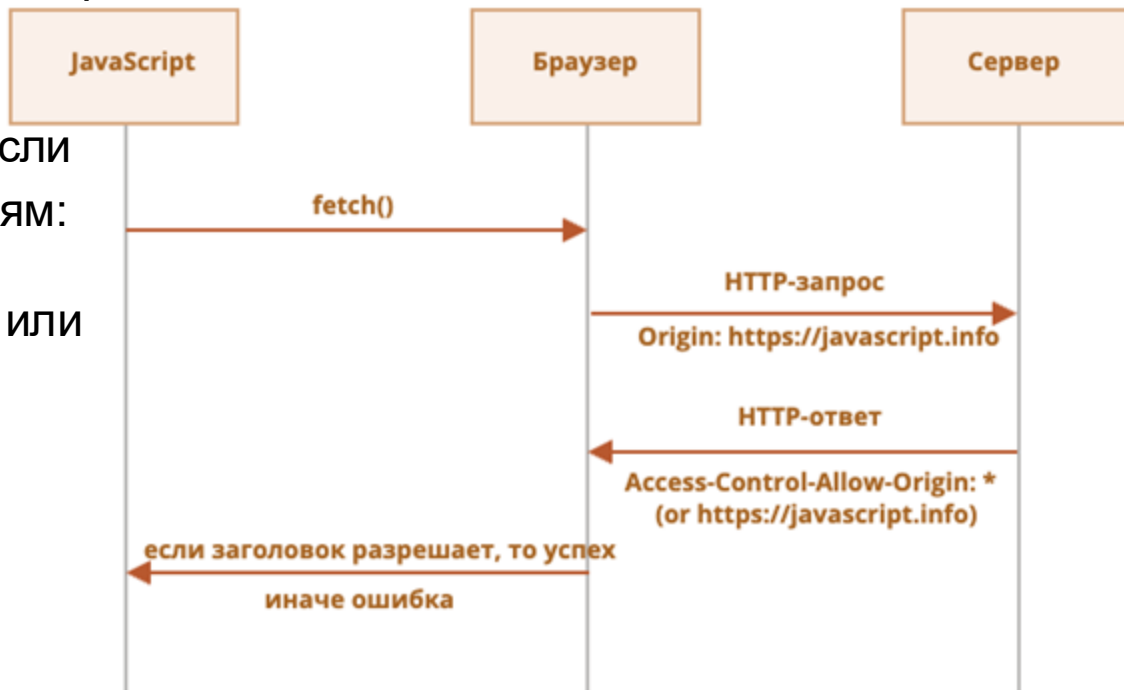
- Он гарантирует, что к запросу на другой источник добавляется правильный заголовок **Origin**.
- Он проверяет наличие разрешающего заголовка **Access-Control-Allow-Origin** в ответе и, если всё хорошо, то JavaScript получает доступ к ответу сервера, в противном случае – доступ запрещается с ошибкой.

```
1 // Находимся на https://evil.com/  
2 const xhr = new XMLHttpRequest();  
3 xhr.open('GET', 'https://e.mail.ru/messages/inbox/', false);  
4 xhr.send();  
5  
6 console.log(xhr.responseText)
```

CORS (простые запросы)

Запросы считаются простыми, если они удовлетворяют двум условиям:

- Простой метод: GET, POST или HEAD
- Простые заголовки
 - Accepts
 - Accept-Language
 - Content-Language
 - Content-Type
 - application/x-www-form-urlencoded
 - multipart/form-data
 - text/plain



1. CORS (запрос сервиса)

Например, мы запрашиваем <https://anywhere.com/request> со страницы <https://javascript.info/page>

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

Ответ сервиса

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

CORS (непростые запросы)

Остальные запросы считаются непростыми. При их отправке нужно понять согласен ли сервер на обработку таких запросов. Эти запросы всегда отсылаются со специальным заголовком Origin.

При отправке непростого запроса, браузер сделает на самом деле два HTTP-запроса.

- «Предзапрос» (английский термин «preflight») OPTIONS. Содержит название желаемого метода в заголовке **Access-Control-Request-Method**, а если добавлены особые заголовки, то и их тоже — в **Access-Control-Request-Headers**.
- Основной HTTP-запрос с заголовком Origin

CORS (непростые запросы)

Предварительный запрос использует метод OPTIONS, который содержит 3 заголовка

- Origin - содержит домен источника
- Access-Control-Request-Method - содержит HTTP метод запроса
- Access-Control-Request-Headers - содержит список HTTP заголовков запроса

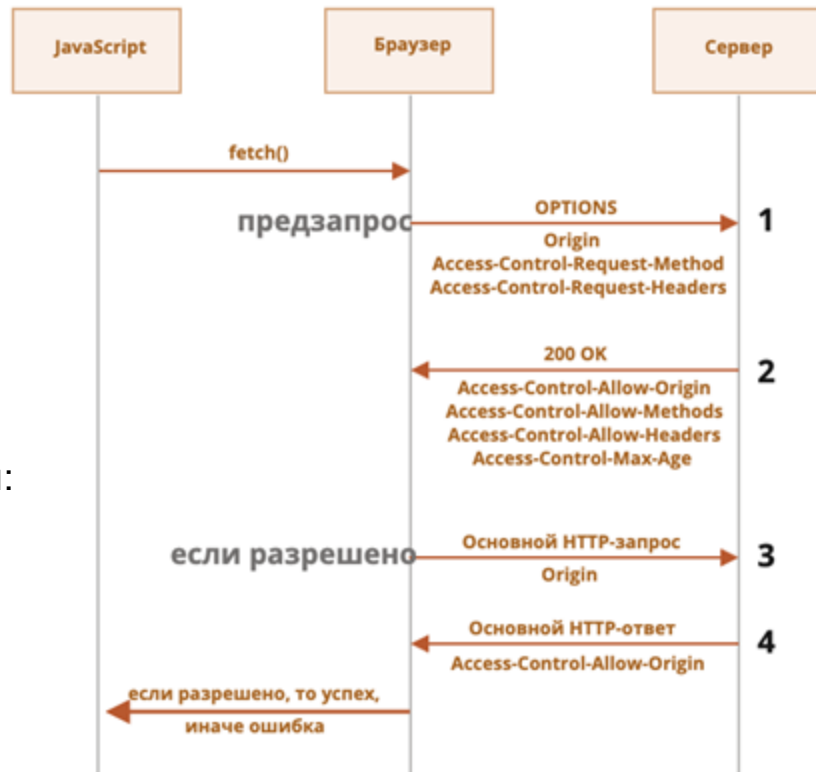
Если сервер согласен - он должен ответить заголовками:

Access-Control-Allow-Origin - список разрешенных источников

Access-Control-Allow-Method - содержит список разрешенных методов

Access-Control-Allow-Headers - содержит список разрешенных заголовков

Access-Control-Max-Age - указывает количество секунд, на которое можно кешировать решение



1. CORS (предзапрос сервиса)

Например, мы запрашиваем <https://site.com/service.json> со страницы <https://javascript.info/page> с методом PATCH и заголовками Content-Type и API-Key

```
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
```

Ответ сервиса на предзапрос

```
1 200 OK
2 Access-Control-Allow-Origin: https://javascript.info
3 Access-Control-Allow-Methods: PUT, PATCH, DELETE
4 Access-Control-Allow-Headers: API-Key, Content-Type, If-Modified-Since, Cache-Control
5 Access-Control-Max-Age: 86400
```

2. CORS (запрос сервиса)

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

Ответ сервиса

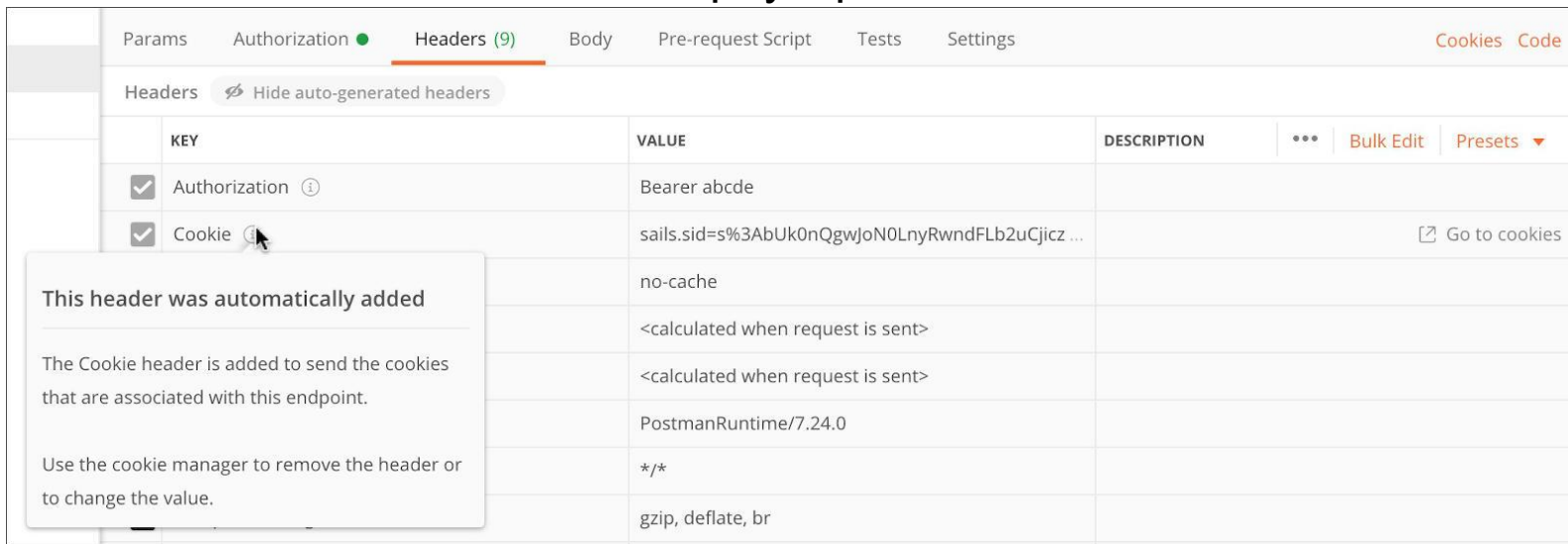
```
1 200 OK
2 Content-Type: text/html; charset=UTF-8
3 Access-Control-Allow-Origin: https://javascript.info
```

Ответ сервиса с
данными авторизации

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

Как хранить данные?

- Кеш браузера - сохраняет только ответы на запросы
- Cookies - подходит для хранения сессий, имеет маленький размер
- Web Storage API - механизм хранения key/value значений (localStorage)
- WebSQL/IndexedDB - база данных в браузере



The screenshot shows the Postman interface with the 'Headers' tab selected. A tooltip is displayed over the 'Cookie' header, stating: 'This header was automatically added. The Cookie header is added to send the cookies that are associated with this endpoint. Use the cookie manager to remove the header or to change the value.'

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Authorization ⓘ	Bearer abcde	
<input checked="" type="checkbox"/> Cookie ⓘ	sails.sid=s%3AbUk0nQgwJoN0LnyRwndFLb2uCjicz ...	Go to cookies
	no-cache	
	<calculated when request is sent>	
	<calculated when request is sent>	
	PostmanRuntime/7.24.0	
	/	
	gzip, deflate, br	

Куки



```
1 GET / HTTP/1.1
2 Host: example.com
```

```
1 HTTP/1.1 200 OK
2 Set-Cookie: name=value
3 Content-Type: text/html
```

```
1 GET / HTTP/1.1
2 Host: example.com
3 Cookie: name=value
```

- **Ку́ки** (*cookie*, букв. — «печенье») — небольшой фрагмент данных, **установленный и отправленный** веб-сервером. **Хранимый** на компьютере пользователя.
- Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта **пересылает** этот фрагмент данных веб-серверу в составе HTTP-запроса.
- Применяется для сохранения данных на стороне клиента (пользователя)

Cookie

- Браузер должен хранить как минимум 4096 байт кук
- Минимум 20 шт. на домен
- Минимум 300 шт. всего
- Имена не чувствительны к регистру

Используется для

- Аутентификация пользователя
- Хранение настроек пользователя
- Отслеживание сеанса (сессия)
- Сбор статистики
- Проведение экспериментов



Подробнее тут <https://learn.javascript.ru/cookie>

Cookie (параметры)

- Expires - дата истечения срока действия куки
- Max-Age - срок действия куки в секундах
- Domain - домен определяет, где доступен файл куки
- Path - урл префикс пути, по которому куки будут доступны
- Secure - куки следует передавать только по HTTPS
- HttpOnly - запрет на получение куки из JavaScript

Samesite — сравнительно новый параметр кук, предоставляющий дополнительный контроль над их передачей согласно Origin policy. Важно заметить, что данная настройка работает только с **secure** cookies.

Возможные значения: Strict, Lax, None.

Cookie (strict)

Cookies с `samesite=strict` **никогда** не отправятся, если пользователь пришел не с этого же сайта.

Важно помнить, что cookies не будут пересылаться также при навигации высокого уровня (т.е. даже при переходе по ссылке)

Пример: vk.com -> site.com -> cookies не передаются