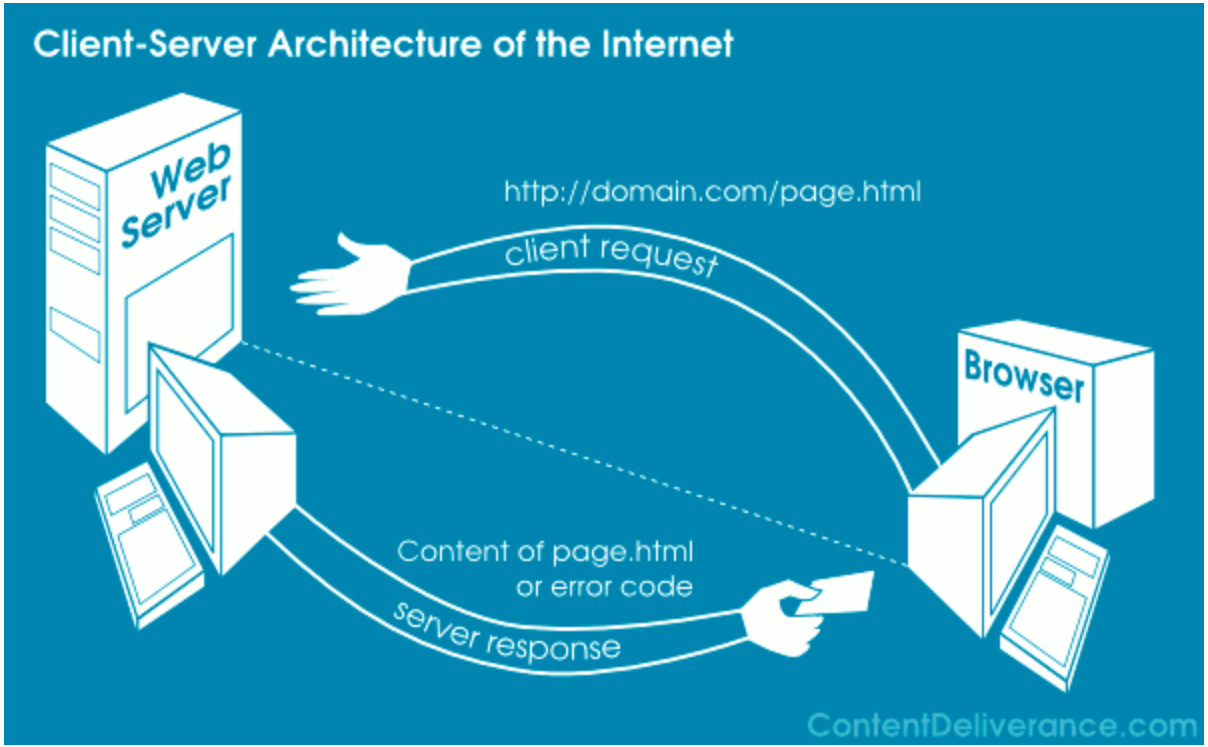
Лекция 6

AJAX

Работа с сетью

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

Протокол HTTP



Ajax

Ajax 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.

Что передаем по сети

- Текстовые данные
- JSON
- Бинарные данные
- Файлы

JSON

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.

```
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "wife": null
}
```

Blob

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

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
- Инициализируем его
- Посылаем запрос
- Слушаем ответ

1. XMLHttpRequest (создаем)

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

2. XMLHttpRequest (инициализируем)

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

3. XMLHttpRequest (посылаем запрос)

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

4. XMLHttpRequest (слушаем ответ)

```
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 (слушаем ответ)

После получения ответа мы можем достать его из xhr

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

XMLHttpRequest (асинхронный запрос)

```
// 1. Создаём новый XMLHttpRequest-объект
let xhr = new XMLHttpRequest();

// 2. Настраиваем его: GET-запрос по URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Отсылаем запрос
xhr.send();

// 4. Этот код сработает после того, как мы получим ответ сервера
xhr.onload = function() {
  if (xhr.status != 200) { // анализируем HTTP-статус ответа, если статус не 200, то произошла ошибка
    alert(`Ошибка ${xhr.status}: ${xhr.statusText}`); // Например, 404: Not Found
  } else { // если всё прошло гладко, выводим результат
    alert(`Готово, получили ${xhr.response.length} байт`); // response -- это ответ сервера
  }
};

xhr.onerror = function() {
  alert("Запрос не удался");
};
```

XMLHttpRequest (синхронный запрос)

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);

try {
  xhr.send();
  if (xhr.status != 200) {
    alert(`Ошибка ${xhr.status}: ${xhr.statusText}`);
  } else {
    alert(xhr.response);
  }
} catch(err) { // для отлова ошибок используем конструкцию try...catch вместо onerror
  alert("Запрос не удался");
}
```

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 отета
- `LOADING = 3` - ответ в процессе передачи
- `DONE = 4` - запрос завершен

```
1 xhr.onreadystatechange = function() {
2   if (xhr.readyState == 3) {
3     // загрузка
4   }
5   if (xhr.readyState == 4) {
6     // запрос завершён
7   }
8 };
9
```

XMLHttpRequest (отмена запроса)

```
1 xhr.abort();
```

XMLHttpRequest (HTTP-заголовки)

```
1 // Устанавливаем заголовок для передачи в запрос
2 xhr.setRequestHeader('Content-Type',
3 'application/json');
4 // Получаем заголовки из запроса
5 xhr.getResponseHeader('Content-Type')
6
7 // Получаем все заголовки из запроса
8 xhr.getAllResponseHeaders()
```

XMLHttpRequest (POST-запрос)

```
<form name="person">
  <input name="name" value="Петя">
  <input name="surname" value="Васечкин">
</form>

<script>
  // заполним FormData данными из формы
  let formData = new FormData(document.forms.person);

  // добавим ещё одно поле
  formData.append("middle", "Иванович");

  // отправим данные
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "/article/xmlhttprequest/post/user");
  xhr.send(formData);

  xhr.onload = () => alert(xhr.response);
</script>
```

XMLHttpRequest (POST-запрос)

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
  name: "Вася",
  surname: "Петров"
});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```

XMLHttpRequest (прогресс отправки)

```
1 xhr.upload.onprogress = function(event) {
2   alert(`Отправлено ${event.loaded} из ${event.total} байт`);
3 };
4
5 xhr.upload.onload = function() {
6   alert(`Данные успешно отправлены.`);
7 };
8
9 xhr.upload.onerror = function() {
10   alert(`Произошла ошибка во время отправки: ${xhr.status}`);
11 };
```

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
- WebSQL/IndexedDB - база данных в браузере

Cookie

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

HTTP is a stateless protocol

Cookie

`Cookie` - небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя. Веб-клиент всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в виде HTTP - запросы

Cookie (спецификация)

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

1. Cookie (клиент -> сервер)

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

2. Cookie (клиент <- сервер)

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

3. Cookie (клиент -> сервер)

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

4. Cookie (клиент -> сервер)

```
1 HTTP/1.1 200 OK
2 Set-Cookie: name=value2
3 Content-Type: text/html
```

5. Cookie (клиент -> сервер)

```
1 GET / HTTP/1.1
2 Host: example.com
3 Cookie: name=value2
```

Cookie (использование)

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

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

Cookie (JavaScript)

```
1 alert( document.cookie ); // cookie1=value1; cookie2=value2;...
2
3 document.cookie = "user=John"; // обновляем только куки с именем 'user'
4 alert(document.cookie); // user=John, cookie1=value1; cookie2=value2;...
```

Cookie (XHR)

```
1 let xhr = new XMLHttpRequest();
2 xhr.withCredentials = true;
3
4 xhr.open('POST', 'http://anywhere.com/request');
```

Same Origin Policy

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

Same Origin Policy

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

Same Origin Policy (источник)

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

Same Origin Policy (одинаковый источник)

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

Same Origin Policy (разные 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

Same Origin Policy (типы взаимодействия с ресурсами)

- Cross Origin “Запись” - обычно разрешена
- Cross Origin “Встраивание” - обычно разрешено
- Cross Origin “Чтение” - по-умолчанию запрещено

Cookie (samesite)

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

CORS

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

CORS (XHR)

```
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 (браузер)

Браузер играет роль доверенного посредника:

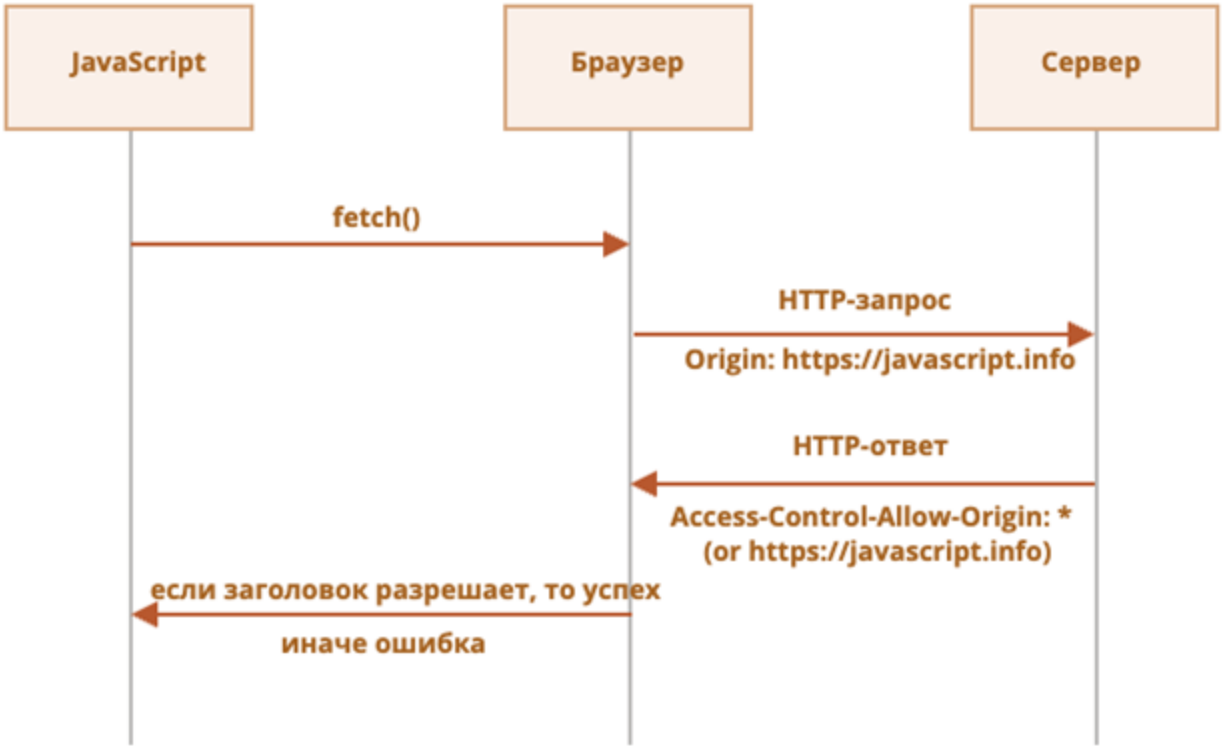
- Он гарантирует, что к запросу на другой источник добавляется правильный заголовок Origin.
- Он проверяет наличие разрешающего заголовка **Access-Control-Allow-Origin** в ответе и, если всё хорошо, то JavaScript получает доступ к ответу сервера, в противном случае – доступ запрещается с ошибкой.

CORS (простые запросы)

Запросы считаются простыми, если они удовлетворяют двум условиям:

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

CORS (простые запросы)



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
```

2. CORS (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.

CORS (непростые запросы)

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 (OPTIONS запрос)

Предварительный запрос использует метод OPTIONS, который содержит 3 заголовка

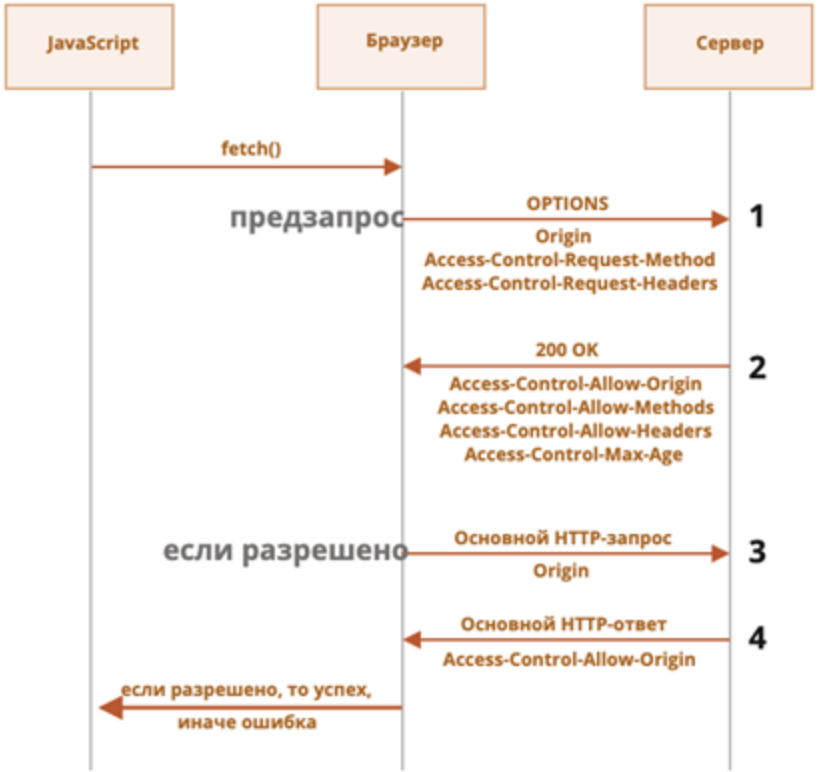
- Origin - содержит домен источника
- Access-Control-Request-Method - содержит HTTP метод запроса
- Access-Control-Request-Headers - содержит список HTTP заголовков запроса

CORS (OPTIONS запрос)

Если сервер согласен принять запрос, то он должен ответить 3 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

CORS (непростые запросы)



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
```

2. CORS (ответ сервиса на предзапрос)

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

3. 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
```

4. CORS (ответ сервиса)

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

CORS (дополнительные заголовки)

```
1 200 OK
2 Content-Type:text/html; charset=UTF-8
3 Access-Control-Allow-Origin: https://javascript.info
4 Access-Control-Expose-Headers: X-Uid, X-Secret
```

CORS (данные 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 еще можно решить проблему корсов?