

ХРАНЕНИЕ СОСТОЯНИЯ НА КЛИЕНТЕ И ОТПРАВКА НА СЕРВЕР



ВЛАДИСЛАВ ВЛАСОВ / DEVELOPER EXPRESS



ВЛАДИСЛАВ ВЛАСОВ

Инженер-программист Девелопер Софт



ihost@mail.ru



fb.me/vladislav.vlasov.7773



ПЛАН ЗАНЯТИЯ

1. [XMLHttpRequest](#) (продолжение)
2. [Cookies](#) (куки)
3. [localStorage](#)
4. [Fetch API](#)

XMLHttpRequest

(ПРОДОЛЖЕНИЕ)

УЧИМСЯ ОТПРАВЛЯТЬ ДАННЫЕ

На предыдущих лекциях мы познакомились с тем, как асинхронно получать данные с сервера с помощью XMLHttpRequest и GET-запросов.

Сегодня мы будем отправлять данные на сервер.



POST-ЗАПРОС

POST-запросы необходимо использовать для добавления каких-либо данных на сервере. Например, если нужно оставить комментарий в блоге или загрузить файл на сервер.

Для передачи этих данных используется тело HTTP-запроса. У GET-запроса не может быть тела, а у POST оно есть.

ПИШЕМ POST-ЗАПРОС

С точки зрения JavaScript POST-запрос отличается от GET несильно:

```
1  const xhr = new XMLHttpRequest()  
2  xhr.addEventListener('load', (e) => {  
3    console.log(xhr.response);  
4  });  
5  xhr.open('POST', '/path');  
6  xhr.send('Привет');
```

Отличия от GET-запроса присутствуют только в последних строчках:

- используем `open('POST')` вместо `open('GET')`;
- тело запроса передается первым аргументом метода `send()`.

ОТПРАВЛЯЕМ ДАННЫЕ ИЗ ФОРМЫ

Для удобного взаимодействия с данными, введенными пользователем в форму, был введен специальный объект — `FormData`.

ОТПРАВЛЯЕМ ДАННЫЕ ИЗ ФОРМЫ

```
1 <form id="reg-form">
2   <input name="name">
3   <input name="age" value="18">
4 </form>
5 <script>
6   const form = document.getElementById('reg-form');
7   const formData = new FormData(form);
8   for (const [k, v] of formData) {
9     console.log(k + ': ' + v);
10  }
11  // name:
12  // age: 18
13 </script>
```

append()

С помощью метода `append()` можно добавлять пары ключ-значение в созданный объект `FormData`:

```
formData.append('key', 'value');
```

ПУСТОЙ ОБЪЕКТ БЕЗ ФОРМЫ

Также можно создавать пустой объект `FormData`, вообще не имея какой-либо формы в HTML и затем наполнять ее через `append()`:

```
1  const formData = new FormData();  
2  formData.append('name', 'Иван');  
3  formData.append('age', '18');
```

ОТПРАВЛЯЕМ ОБЪЕКТ НА СЕРВЕР

После создания и заполнения формы значениями ее можно отправить на сервер с помощью `XMLHttpRequest`, если ссылку на объект `FormData` передать первым параметром метода `send()`:

```
1  const xhr = new XMLHttpRequest();  
2  // ...  
3  xhr.send(formData);
```

ТЕЛО ОТПРАВЛЕННОГО ОБЪЕКТА

В этом случае на сервер будет отправлен *multipart-запрос*, тело которого будет следующим:

```
-----WebKitFormBoundaryQAWcWWA9tNQjAwje  
Content-Disposition: form-data; name="name"
```

Иван

```
-----WebKitFormBoundaryQAWcWWA9tNQjAwje  
Content-Disposition: form-data; name="age"
```

18

```
-----WebKitFormBoundaryQAWcWWA9tNQjAwje--
```

УСЛОВИЯ ОТПРАВКИ JSON

JSON — наиболее популярный формат передачи данных между сервером и клиентом в веб-приложениях.

Для отправки JSON с помощью POST-запроса необходимо следующее:

1. вручную установить заголовок запроса `Content-Type: application/json`;
2. сериализовать объект в строку с помощью `JSON.stringify()`.

ОТПРАВЛЯЕМ JSON

```
1  const xhr = new XMLHttpRequest();  
2  // ...  
3  xhr.open('POST', '/json');  
4  xhr.setRequestHeader('Content-Type', 'application/json');  
5  xhr.send(JSON.stringify({ name: 'Иван' }));
```

ВАЖНЫЙ МОМЕНТ

Метод `setRequestHeader()` может быть вызван только после вызова `open()`, но перед вызовом метода `send()`.

JSON ВНУТРИ FormData

С другой стороны, можно отправить JSON внутри уже известного объекта `FormData`. Например, вот так:

```
1  const data = { name: 'Иван', age: 18 };
2  const formData = new FormData();
3  formData.append('data', JSON.stringify(data));
```

ДРУГИЕ HTTP-МЕТОДЫ

Методы, используемые для изменения данных на сервере:

- PUT – `open('PUT')` – для создания (или полной перезаписи) какого-то ресурса;
- DELETE – `open('DELETE')` – для удаления ресурса;
- PATCH – `open('PATCH')` – для перезаписи какой-то части ресурса (не всего целиком).

Для всех этих HTTP-методов доступно тело запроса.



COOKIES (КУКИ)



ПРИЧИНА ПОЯВЛЕНИЯ COOKIES

Проблема: сервер хочет принимать данные только от авторизованных пользователей. Как это проверить?

Для решения этой задачи и были придуманы куки.



Куки – это пары строк ключ-значение, которые сохраняются даже после закрытия браузера.





ПОЛЬЗОВАТЕЛЬ = ИДЕНТИФИКАТОР

Для идентификации конкретного пользователя сервер записывает в куки специальный идентификатор (строку или число), которые определяют сессию и соответственно конкретного пользователя для сервера. А затем вместе с каждым запросом браузер отправляет на сервер куки автоматически.

УСТАНОВКА

Устанавливать куки можно либо с помощью JavaScript-кода, либо со стороны сервера с помощью HTTP-заголовка `Set-Cookie`:

```
Set-Cookie: sessionId=XXXyyyZZZ
```

document.cookie

Для работы с куками силами JavaScript в браузере существует специальный BOM-объект `document.cookie`. Через него можно как устанавливать новые значения, так и читать существующие.

ЗАПИСЫВАЕМ НОВУЮ КУКУ

Запись в куки осуществляется с помощью присваивания в `document.cookie`. Причем такое присваивание не перезаписывает старые куки, а добавляет новую.

Новую куку нужно записывать в виде одной строки, где ключ отделен от значения знаком `=`:

```
document.cookie = 'firstname=Иван';  
document.cookie = 'lastname=Петров';
```

ПРЕДОБРАБОТКА

Если значение содержит пробелы, точки с запятой или запятые, то его необходимо предобработать с помощью функции

`encodeURIComponent()`:

```
document.cookie = 'user=' +  
    encodeURIComponent('Иван Иванович Иванов; 1945 г.');
```

ПОЛУЧАЕМ COOKIES

А вот прочитать можно только сразу все куки, которые установлены на сайте. Это одна большая строка, в которой куки разделены между собой точкой с запятой ;:

```
console.log(document.cookie);  
// firstname=Иван; lastname=Петров
```



ФУНКЦИИ-ПОМОЩНИКИ

Это достаточно парадоксально. Куки — это пары ключ-значение, но прочитать их можно только все сразу в виде одной строки. Для более удобного работы с куками лучше использовать дополнительные функции-помощники (или целую библиотеку) для чтения конкретной куки и создания новой (об этом далее).

ОПЦИИ ДЛЯ COOKIES

На самом деле, у куки кроме значения может быть масса дополнительных параметров, которые можно указать при создании куки:

```
document.cookie = 'name=value;  
Expires=Mon, 01 May 2018 21:41:37 GMT; Path=/api;  
Domain=.mysite.com';
```

ОСНОВНЫЕ ПАРАМЕТРЫ

- **Expires** — определяет время жизни куки. Если не указывать, то кука исчезнет после закрытия браузера.
- **Path** — для какого пути (и всех путей, содержащих указанное значение) кука автоматически подставится в запрос. Если не указано — значит, берется текущий путь.
- **Domain** — домен, на котором доступна кука. Можно указывать текущий домен или поддомены. Если не указано, то будет взят текущий домен. Если, как в примере, **Domain=.mysite.com** (с точкой в начале), то кука установлена для домена и всех поддоменов.

УДАЛЯЕМ COOKIE

Удалить куку явно нельзя, но можно установить ей дату окончания существования в далекое прошлое, и браузер удалит такую куку автоматически:

```
document.cookie = 'name=;  
Expires=Thu, 01 Jan 1970 00:00:00 GMT';
```

В этом случае значение можно вообще не указывать.



ОГРАНИЧЕНИЯ

Все браузеры накладывают на куки ограничения по размеру, как на размер отдельной куки, так и на общее количество:

- имя и значение не должны превышать **4кб**;
- общее количество кук на домен имеет ограничение — не более **50**. В отдельных браузерах еще меньше.



LOCALSTORAGE



***localStorage** – это специальный объект BOM, с помощью которого можно долговременно хранить информацию на клиенте.*

Для каждого домена свое собственное хранилище.

Для доступа к данным предоставляется удобный интерфейс для чтения и записи пар ключ-значение (строки).





ОБЛАСТИ ПРИМЕНЕНИЯ

Обычно localStorage используют для следующего:

- Хранение идентификаторов пользователя/сессии, персональной информации.
- Удобство пользователя: для сохранения какой-то информации, которую пользователь ввел, но еще не отправил на сервер.
- Быстродействие: если есть информация в localStorage, то не нужно делать лишний запрос на сервер.



ОГРАНИЧЕНИЯ РАЗМЕРА

Общий размер памяти под localStorage обязательно ограничен браузером.

Современные НЕ мобильные браузеры предоставляют до **10 Мб** под localStorage для каждого домена, мобильные обычно меньше.

ЗАПИСЫВАЕМ ДАННЫЕ

Давайте начнем с записи в localStorage. Это можно сделать двумя различными способами — с помощью метода `setItem()` или непосредственно с помощью свойства объекта:

```
1 localStorage.setItem('lastname', 'Иванов');  
2 // или как со свойством объекта  
3 localStorage['lastname'] = 'Петров';  
4 // можно и без квадратных скобок  
5 localStorage.lastname = 'Сидоров';
```

ЧИТАЕМ ДАННЫЕ

Теперь даже после перезагрузки страницы или браузера целиком фамилию можно будет прочитать.

Снова можно прочитать значение либо через метод `getItem()`, либо через свойство объекта:

```
1 let lastname = localStorage.getItem('lastname');  
2 console.log(lastname); // 'Сидоров'  
3  
4 // или через чтения свойства объекта  
5 console.log(localStorage.lastname); // 'Сидоров'
```



length

У объекта `localStorage` есть еще полезное свойство `length` — это общее количество ключей, которые определены в данный момент.

ПОЛНОСТЬЮ ОЧИЩАЕМ ХРАНИЛИЩЕ

`localStorage.clear()` полностью очищает текущее хранилище. Например, нужно вызывать во время разлогинивания с сайта, чтобы удалить все связанные с пользователем данные.

УДАЛЯЕМ ОТДЕЛЬНЫЙ КЛЮЧ

Кроме полной очистки хранилища, можно удалить конкретный ключ и связанные с ним значения с `removeItem()` или с помощью оператора `delete`:

```
localStorage.removeItem('user');  
delete localStorage.user;
```

РАБОТАЕМ С ОБЪЕКТАМИ

А что, если нужно записать объект?

Для этого необходимо использовать сериализацию/десериализацию с помощью `JSON.stringify()` и `JSON.parse()` соответственно;

```
1 function saveUser(user) {  
2     localStorage.user = JSON.stringify(user);  
3 }  
4  
5 function getUser() {  
6     JSON.parse(localStorage.user);  
7 }
```

ЛОВИМ ОШИБКИ

Причем подобная реализация `getUser()` не очень безопасная, потому что вызов `JSON.parse()` с любыми некорректными данными приведет к исключению.

Поэтому при чтении объектов из `localStorage` лучше всегда использовать `try-catch`:

```
1 function getUser() {  
2   try {  
3     return JSON.parse(localStorage.getItem('user'));  
4   } catch (e) {  
5     return null;  
6   }  
7 }
```

ПЕРЕПОЛНЕНИЕ ХРАНИЛИЩА

Если постоянно записывать новые данные в localStorage, то можно достичь того самого ограничения общего размера. Или этого можно достичь разовой записью, например, вот так:

```
1 localStorage.bigdata = new Array(1e7).join('x')
2 // Uncaught DOMException:
3 // Failed to set the 'bigdata' property on 'Storage':
4 // Setting the value of 'bigdata' exceeded the quota.
```

Не повторяйте этого!

ПОДПИСЫВАЕМСЯ НА СОБЫТИЯ

Можно подписаться на событие, которое возникает во время изменения содержимого хранилища. Это событие с названием `storage` у объекта `window`:

```
1 window.addEventListener('storage', () => {  
2   console.log(e.key, e.oldValue, e.newValue);  
3 });  
4  
5 localStorage.x = 1      // 'x', null, '1'  
6 localStorage.x = 2      // 'x', '1', '2'  
7 delete localStorage.x   // 'x', '2', null  
8 localStorage.clear()    // null, null, null
```



СРАВНЕНИЕ ИНТЕРФЕЙСОВ ХРАНЕНИЯ

Сравним localStorage, sessionStorage и Cookies: все эти интерфейсы предоставляют возможность хранить дополнительную данные на клиенте в рамках одного домена, но давайте подведем итог по их различиям.

СРАВНЕНИЕ: COOKIES

- (плюс) могут быть установлены с сервера (заголовок `Set-Cookie`);
- (плюс) могут быть недоступны на клиенте (опция `HttpOnly`);
- (плюс) имеют настройку времени жизни (опция `Expires`);
- (плюс/минус) автоматически отправляются на сервер (в зависимости от опции `Path`);
- (минус) существенные ограничения на количество/размер одной куки/общий размер;
- (минус) неудобный интерфейс.



СРАВНЕНИЕ: LOCALSTORAGE

- (плюс) ограничен только общий размер хранимых данных;
- (плюс) удобный интерфейс;
- (плюс/минус) нет автоматической отправки на сервер;
- (минус) значения не могут быть установлены с сервера.



СРАВНЕНИЕ: SESSIONSTORAGE

- обладает всеми свойствами localStorage, кроме времени жизни (до завершения работы браузера).



FETCH API

НОВЫЙ СПОСОБ ОТПРАВКИ AJAX-ЗАПРОСОВ

Fetch API и глобальная функция `fetch` — это новый способ осуществления запросов, основанный на `Promise`.

НАЧИНАЕМ С ПРОСТОГО

Простой GET-запрос:

```
1  fetch('/resource')
2    .then((res) => {
3      // обрабатываем ответ
4    })
5    .catch((err) => {
6      // обрабатываем ошибку
7    });
```

УСЛОЖНЯЕМ ЗАДАЧУ

Теперь давайте попробуем что-то посложнее. Для дополнительной настройки запроса мы будем использовать второй необязательный параметр:

```
1  const request = fetch('/resource', {  
2    body: JSON.stringify({ name: 'Иван' })  
3    credentials: 'same-origin' // 'include' | 'omit'  
4    method: 'POST',  
5    headers: {  
6      'Content-Type': 'application/json'  
7    },  
8  });
```

ДОСТУПНЫЕ ПАРАМЕТРЫ

- `body` – собственно тело запроса;
- `credentials` – устанавливает режим работы с cookie;
- `method` – HTTP-метод;
- `headers` – для установки HTTP-хидеров для запроса.

ПОЛУЧАЕМ И ОБРАБАТЫВАЕМ ОТВЕТ

Для обработки кодов ответа и получение ответа в форме JSON нужно:

```
1  fetch(/* ... */)
2    .then((res) => {
3      if (200 <= res.status && res.status < 300) {
4        return res;
5      }
6      throw new Error(response.statusText);
7    })
8    .then((res) => { return res.json(); })
9    .then((data) => { /* обрабатываем данные */ })
10   .catch((error) => { /* обрабатываем ошибку */ });
```

СИЛЬНАЯ СТОРОНА `fetch`

Сильная сторона `fetch` — это промисы, потому что их удобно комбинировать последовательно или параллельно:

```
1  fetch('/resource1')
2    .then(() => fetch('/resource2'))
3    .then(() => fetch('/resource3'))
4    .then(() => {
5      // ...
6    })
7
8  Promise.all([
9    fetch('/resource1'), fetch('/resource2')
10  ]).then(([res1, res2]) => {
11    // ...
12  })
```


ВАЖНО ПОМНИТЬ

- запрос-промис не переходит в состояние `rejected` при ошибочных кодах ответа (4xx, 5xx);
- не происходит автоматическая отправка/прием `cookies` (необходимо явно указывать опцию `credentials`);
- для работы в старых браузерах нужно использовать [полифилл](#).



ИТОГИ

XMLHttpRequest

- Для отправки данных на сервер используйте POST-запрос.
- У POST-запроса есть тело, внутри которого передаются данные.
- Тело запроса передается первым аргументом метода `send()`.
- `FormData` – объект для работы с пользовательскими данными из форм.
- Метод `append()` добавляет пары ключ-значение в объект `FormData`.
- Допустимо создавать пустой объект `FormData`.
- Передайте ссылку на объект `FormData` в методе `send()` первым параметром и данные уйдут на сервер.

JSON

- Для отправки JSON через POST-запрос установите заголовок запроса `Content-Type: application/json` и сериализуйте объект с помощью `JSON.stringify()`.
- Метод `setRequestHeader()` вызывайте строго после `open()`, но до `send()`.
- JSON можно отправить внутри `FormData`.
- `open('PUT')` создает новый или перезаписывает полностью существующий ресурс.
- `open('PATCH')` частично перезаписывает ресурс.
- `open('DELETE')` удаляет ресурс.

COOKIES

- Cookies нужны для хранения данных на клиентской стороне.
- Для работы с куками есть объект `document.cookie`.
- Кука записывается в виде одной строки, где ключ и значение разделены знаком равно. При этом новая кука не переписывает предыдущее значение, а добавляется как новая.
- Если в строке встречаются пробелы, знаки препинания, то используйте `encodeURIComponent()`.
- Куки читаются только все сразу в виде строки с разделителем `;`.

COOKIES: ПАРАМЕТРЫ И ОГРАНИЧЕНИЯ

- Параметры: `Expires` определяет время жизни; `Path` указывает, куда подставляться кука; `Domain` указывает домен, на котором кука доступна.
- Куки не удаляются, только при помощи установку даты окончания существования в прошлом.
- Куки ограничены. Одна кука не больше 4кб, общее количество - не более 50 на домен.

LOCALSTORAGE

- `localStorage` – долговременное хранилище информации на клиенте.
- Хранилище ограничено по размеру – максимум 10 Мб.
- У каждого домена свой собственный `localStorage`.
- Данные записываются через `setItem()` или добавлением свойства.
- Читаются при помощи `getItem()` или обращением к свойству.
- `localStorage.length` показывает число существующих ключей.

LOCALSTORAGE

- `localStorage.clear()` очищает хранилище.
- Частичная очистка при помощи `removeItem()` или оператора `delete`.
- При записи/извлечении объекта используйте `JSON.stringify()` и `JSON.parse()`.
- Не забывайте отлавливать ошибки при помощи `try-catch`.
- У объекта `window` есть событие `storage`.

FETCH API

- `fetch` – глобальная функция.
- `body` содержит тело запроса.
- `credentials` отвечает за режим работы с cookies.
- `method` – HTTP-метод.
- `headers` для установки хедеров запроса.
- Промисы удобно использовать для последовательных или параллельных запросов.
- Запрос-промис не переходит в состояние `rejected` при ошибках.
- Нет автоматической отправки куков на сервер.
- Если нужны старые браузеры, то используйте [полифилл](#).

СВОЙСТВО `state`

- Свойство `state` только для чтения, если вы хотите изменить текущее состояние, то необходимо использовать метод `setState`.
- React следит за текущим состоянием и при каждом его изменении вызывает метод `render`, который перерисовывает компонент.
- Так как при изменении состояния вызывается метод `render`, то вызов в нём `setState` приведет к бесконечному циклу.
- Следует держать состояние компонента в чистоте и вызывать метод `setState` только по необходимости.



НЕТОЛОГИЯ
университет интернет-профессий

Задавайте вопросы и напишите отзыв о лекции!

ВЛАДИСЛАВ ВЛАСОВ

 ihost@mail.ru

 fb.me/vladislav.vlasov.7773