нетология

Конспект по теме 3.2 «Хранение состояния на клиенте и отправка на сервер»

Содержание конспекта:

1. Работа с LocalStorage
2. JSON
3. Cookies и использование document.cookie

1. Работа с LocalStorage

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

Каждому домену предоставляется собственное хранилище. Это сделано для безопасности, чтобы один сайт не мог прочитать данные, относящиеся к другому сайту. Специальный объект браузерной модели **LocalStorage** предоставляет интерфейс для длительного хранения информации в виде пар «ключ-значение». Важно, что и ключи, и значения являются строками.

Обычно LocalStorage используют для:

- Хранения идентификаторов пользователя/сессии, персональной информации;
- Для удобства пользователя: хранение какой-то информации, которую пользователь ввёл, но ещё не отправил на сервер.
- Для быстродействия: если есть информация в LocalStorage, то не нужно делать лишний запрос на сервер.

Запись в LocalStorage

Познакомимся с интерфейсом **LocalStorage**. Записывать в него данные можно двумя способами:

1. С помощью метода **setItem**(), который принимает 2 аргумента — ключ и значение:

```
localStorage.setItem('lastname', 'Иванов');
```

2. Непосредственно с помощью свойства объекта, путём обращения к свойству, как в случае с обычным объектом JavaScript:

```
localStorage.lastname = 'Сидоров';
```

Чтение в LocalStorage

Для чтения сохранённых значений из LocalStorage также доступны 2 варианта:

1. Чтение данных из LocalStorage метод **getItem**(), который принимает 1 аргумент — ключ:

```
let lastname = localStorage.getItem('lastname');
console.log(lastname);   // 'Сидоров'
```

2. Чтение данных из LocalStorage через свойство объекта, с помощью обращения к LocalStorage как к объекту JavaScript:

```
console.log(localStorage.lastname);   // 'Сидоров'
```

Кроме того, объект LocalStorage имеет полезное свойство **length**, в котором хранится текущее количество ключей, сохранённых в LocalStorage.

Удаление записей

Для удаления записей в LocalStorage тоже доступны различные варианты:

1. Удаление данных из LocalStorage методом **clear**():

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

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

2. Удаление данных из LocalStorage оператором **delete**:

```
delete localStorage.user;
```

Можно удалять записи поштучно по соответствующему ключу. Для этого можно использовать метод **removeItem**() или стандартный оператор удаления ключа из объекта в JavaScript — оператор **delete**.

Долговременные хранилища, как правило, имеют ограничения по размеру хранимой информации. Современные десктопные браузеры предоставляют до 10 МБ места на жёстком диске для данных одного домена. Мобильные обычно ограничивают размер ещё строже.

Раз размер данных ограничен, значит этого ограничения можно достичь. Например, если мы попытаемся записать очень длинную строку в какой-то ключ, то выбросится исключение.

Переполнение поля LocalStorage одной записью:

```
localStorage.bigdata = new Array(1e7).join('x');

// Uncaught DOMException:
// Failed to set the 'bigdata' property on 'Storage':
// Setting the value of 'bigdata' exceeded the quota.
```

2. JSON

Рассмотрим формат представления и хранения данных JSON и области его применения. Этот формат был придуман специально для того, чтобы сохранять JavaScript-объекты, но сейчас его используют по всему миру вне зависимости от языка программирования.

JSON — текстовый формат обмена и хранения данных, основанный на JavaScript.

При взаимодействии между клиентом и сервером или при записи данных на жёсткий диск мы не можем использовать непосредственно JavaScript-объекты или массивы, однако в этих целях нам доступно использование строк.

Пример сохранения объектов в LocalStorage

Сначала мы должны превратить объект в строку, а уже затем можем его записать в хранилище. Такой процесс преобразования называется **сериализация**.

Сериализация — это процесс преобразования структуры данных в последовательность байтов или строку.

Есть и обратный процесс, который позволяет получить из строки объект JavaScript, он называется десериализацией.

Десериализация — это процесс преобразования строки или последовательности байтов в структуру данных языка программирования.

Для работы с форматом документа JSON в JavaScript существует глобальный объект JSON с 2 методами:

1. **stringify** — для сериализации объектов в строки.
2. **parse** — для десериализации строк обратно в объекты.

```
function saveUser(user) {
   localStorage.user = JSON.stringify(user);
 }

function getUser() {
  return JSON.parse(localStorage.user);
}
```

Причём подобная реализация **getUser**() не безопасная, потому что вызов JSON.parse() с любыми некорректными данными приведёт к исключению. Поэтому при чтении объектов из localStorage лучше всегда использовать try-catch:

```
function getUser() {
  try {
    return JSON.parse(localStorage.getItem('user'));
  } catch (e) {
    return null;
  }
}
```

Переполнение

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

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

События на изменения LocalStorage

Можно подписаться на событие, которое возникает во время изменения содержимого LocalStorage. Это событие с названием **storage** у объекта window. Событие не будет возникать на той же вкладке (или странице, если у браузера нет вкладок), где вносятся изменения, т. к. оно является способом для других вкладок на том же домене использовать хранилище для синхронизации любых внесённых изменений.

Рассмотрим пример, где пользователь, у которого сайт открыт сразу в нескольких вкладках, поменял тему со светлой на тёмную. Сделаем так, чтобы в остальных вкладках тема тоже изменилась:

```
// 1) при изменении пользователем темы на сайте выполнилась строка скрипта:
localStorage.theme = "dark";

// 2) в другой вкладке сработал следующий код:
window.addEventListener('storage', (e) => {
  console.log(e.key, e.oldValue, e.newValue); // будет выведено: "theme", "dark",
"light"
  document.body.classList = [e.newValue];

// 3) проставим класс темы для body
```

});

Итоги

Объект LocalStorage

- Является специальным объектом BOM, с помощью которого можно долговременно хранить информацию на клиенте;
- Доступен в домене, на котором установлен;
- Ограничен только общий размер хранимых данных;
- Имеет удобный интерфейс для работы;
- Не имеет автоматической отправки на сервер.

3. Cookies и использование document.cookie

Ещё на заре интернета сервера не умели идентифицировать пользователей, посещающих веб-страницу, что вызывало определённые сложности при индивидуализации рабочей сессии пользователя. Для решения этой проблемы и были придуманы Cookie.

Cookie (куки) — это пары строк ключ-значение, которые отправляются при каждом HTTP-запросе на сервер, помогая ему понять, какой именно пользователь работает с сайтом сейчас, сохраняя это значение даже после закрытия пользователем браузера.

Для идентификации конкретного пользователя сервер записывает в куки специальный идентификатор (строку или число), который определяет сессию и конкретного пользователя для сервера. Затем вместе с каждым запросом браузер автоматически отправляет на сервер куки.

Использование document.cookie

Так как куки вошли в использование до активного применения JavaScript для написания веб-приложений, то устанавливать куки можно не только с помощью JavaScript, но и со стороны сервера с помощью HTTP-заголовка Set-Cookie.

Установка Cookie при помощи заголовка Set-Cookie:

```
Set-Cookie: sessionid=XXXyyyZZZ
```

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

Запись в куки осуществляется с помощью присваивания в **document.cookie**. Причём такое присваивание не перезаписывает старые cookie, а добавляет новую. Новую куки нужно записывать в виде одной строки, где ключ отделён от значения знаком **=** .

Запись значения в Cookie:

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

Интерфейс записи новой куки может показаться слегка необычным. Мы как бы присваиваем в document.cookie строку, содержащую имя и значение куки,

разделённые знаком равно. Но эта запись не перезаписывает все куки, а добавляет новую.
Не все символы можно использовать в качестве значений куки. Если вы не контролируете установленное значение (например, вводит пользователь) или значение содержит пробелы, точки с запятой или запятые, то его необходимо предобработать с помощью функции **encodeURIComponent**():

Кодирование значения Cookie:

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

Получение значений Cookie:

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

Прочитать можно только сразу все куки, которые установлены на сайте. Это одна большая строка, в которой куки разделены между собой знаком **;**. Что достаточно парадоксально. Куки — это пары ключ-значение, но прочитать их можно только все сразу в виде одной строки. Для более удобной работы с куки лучше использовать дополнительные функции-помощники (или целую библиотеку) для чтения конкретной куки и создания новой.

Рассмотрим функцию, которую можно использовать для получения значения куки по имени:

```
const getCookie = (name) => {
  const value = "; " + document.cookie;
  let parts = value.split("; " + name + "=");
  if (parts.length === 2) {
    return parts
      .pop()
      .split(";")
      .shift();
  }
}
```

В этой функции в строке 3 строка, которая представляет собой куки и их значения с добавленной **';'** в конце, разбивается на 2 части функцией **split**. Разделителем **split** служит **;name=** , где name — имя куки.
В строке 4 проверяется, разделилась ли строка на 2, т. е. встретился ли в ней разделитель **;name=**. Если нет, то нет такой куки. Если да, то происходит следующее:

- parts.pop() возвращает 2-ю часть строки, где как раз содержится значение искомой куки вместе с другими (стр. 6);

- split(";") — полученная подстрока разбивается на отдельные куки разделителем ';' (стр. 7);
- shift() — берётся первый элемент из получившегося массива куки (стр. 8), где содержится значение искомой куки. При этом имя куки отсутствует, т. к. оно служило разделителем.

Различные опции для куки

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

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

- Expires — определяет время жизни куки. Если не указывать, то куки исчезнет после закрытия браузера;
- Path — для какого пути (и всех путей, начинающихся на этот путь) куки будет автоматически подставляться в запрос. Если не указано, значит, берётся текущий путь;
- Domain — домен, на котором доступна куки. Можно указывать текущий домен или поддомены. Если не указано, то будет взят текущий домен. Если как в примере (с точкой в начале), то куки установлена для домена и всех поддоменов.

Удаление куки

Удалить куки явно нельзя, но можно установить ей дату окончания существование в далёкое прошлое, и браузер удалит такую куку автоматически, например:

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

В этом случае значение можно вообще не указывать.

Ограничения куки

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

- Имя и значение не должны превышать 4 КБ;
- Общее количество куки на домен имеет ограничение — не более 50. В некоторых браузерах ещё меньше.

Задача: установить куки с названием user со значением «Вася Пупкин» с истекающей датой 1 Мая 2019.

Варианты:
1. document.cookie = 'user=' + encodeURIComponent("Вася Пупкин") + "; Expires=Mon, 01 May 2019 00:00:00 GMT;'
2. document.setCookie = 'user=' + encodeURIComponent("Вася Пупкин") + "; Expires=Mon, 01 May 2019 00:00:00 GMT;'
3. document.cookie = 'user="Вася, Пупкин"; Expires=Mon, 01 May 2018 00:00:00 GMT;"

Верны варианты 1 и 3. Обратите внимание, в варианте 3 значение будет сохранено вместе с кавычками.

Итоги

Cookies
- Представляют собой пары строк ключ-значение, которые сохраняются даже после закрытия браузера (хранятся как файлы в папке с настройками браузера);
- Доступны в домене (поддомене), на котором установлены (можно настраивать);
- Могут быть установлены с сервера (заголовок Set-Cookie);
- Имеют настройку времени жизни (опция Expires);
- Автоматически отправляются на сервер (в зависимости от опции Path);
- Имеют существенные ограничения на количество, размер одной куки, общий размер.

Сравнение LocalStorage, sessionStorage, cookie

Cookies:

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

LocalStorage:

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

sessionStorage:

- + обладает всеми свойствами LocalStorage, кроме времени жизни — до завершения работы вкладки или окна браузера.

Кросс-доменные запросы, CORS, XHR.withCredentials

Механизм CORS для кросс-доменных запросов

Представим, что нам необходимо запросить данные о залогированном пользователе для нашего веб-приложения по учёту финансов (пусть он будет расположен по адресу www.financeManager.ru) с другого сайта, например, www.example.ru

```
const invocation = new XMLHttpRequest();
const url = 'http://www.example.ru/public-data/';

function getData() {
  if (invocation) {
    invocation.open('GET', url, true);
    invocation.onreadystatechange = handler;
    invocation.send();
```

```
  }
}
```

Но вместо данных в консоли мы видим ошибку:

Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at www.example.ru

Получается ошибка, поскольку в целях безопасности браузеры ограничивают кросс-доменные запросы, инициируемые скриптами. Например, XMLHttpRequest (и Fetch API) следуют политике одного источника (same-origin policy). Это значит, что веб-приложения, использующие такие API, могут запрашивать HTTP-ресурсы только с того домена, с которого были загружены, пока не будут использованы CORS-заголовки.

Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность веб-приложению, работающему на одном домене, получить доступ к выбранным ресурсам с сервера на другом источнике (домене).

Говорят, что агент пользователя (браузер) делает кросс-доменный запрос, если источник текущего документа отличается от запрашиваемого ресурса доменом, протоколом или портом.
С точки зрения безопасности разработчики стандарта XMLHttpRequest предусмотрели все возможные варианты, при помощи которых злоумышленники могли бы сломать какой-нибудь сервер, работающий по старому стандарту и не ожидающий новых видов запросов и заголовков.

Простые и предварительно проверяемые (preflighted) запросы

С точки зрения CORS существует два вида запросов:

- простые,
- предварительно проверяемые.

Простыми считаются запросы, удовлетворяющие следующим условиям:

1. Методы: GET, POST или HEAD;
2. Заголовки из списка:
   - Accept,
   - Accept-Language,
   - Content-Language,

- Content-Type со значением application/x-www-form-urlencoded, multipart/form-data или text/plain.

Предварительно проверяемыми считаются все остальные запросы.

Разница между ними заключается в том, что простой запрос можно сформировать и отправить на сервер и без XMLHttpRequest, например, при помощи HTML-формы. Поэтому сервера на старых технологиях (без поддержки XMLHttpRequest) уже должны были предусмотреть эту потенциальную угрозу. Запросы с нестандартными заголовками или с методом DELETE при помощи отправки формы не создать, поэтому старый сервер может быть к ним не готов и повести себя небезопасным образом. Для этого и появились предварительно проверяемые запросы, где до основного запроса идёт запрос на проверку готовности сервера.

CORS для простых запросов

Вернемся к примеру выше. Предположим, с нашего сайта www.financeManager.ru мы хотим получить данные о пользователе с сайта www.example.ru. Запрос удовлетворяет условиям с прошлого слайда и поэтому является простым запросом.

```
const invocation = new XMLHttpRequest();
const url = 'http://www.example.ru/public-data/';

function getData() {
  if (invocation) {
    invocation.open('GET', url, true);
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```

Посмотрим на заголовки запроса браузера:

```
GET /resources/public-data/ HTTP/1.1
Host: www.example.ru
Referer: www.financeManager.ru
Origin: http://www.financeManager.ru
...
```

Самый важный заголовок, на который стоит обратить внимание, — это Origin, показывающий домен, с которого осуществлён запрос.

Посмотрим теперь на заголовки ответа с сервера http://www.example.ru:

```
HTTP/1.1 200 OK
Date: Fri, 01 Feb 2019 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
Content-Type: application/xml
...
```

Как мы видим, в ответ сервер отправляет заголовок Access-Control Allow-Origin, в строке 4. В нашем случае сервер отвечает Access Control-Allow-Origin: * , что означает, что к ресурсу может обращаться любой домен в межсайтовом режиме. Если владельцы http://www.example.ru хотят ограничить доступ к ресурсу только запросами с http://www.finance Manager.ru, они отправят обратно:

```
Access-Control-Allow-Origin: http://www.financeManager.ru
```

Теперь ни один домен, кроме http://www.financeManager.ru (идентифицируемый заголовком ORIGIN в запросе, как в строке 4 выше), не может получить доступ к ресурсу с помощью кросс-доменного запроса. Заголовок Access-Control Allow-Origin должен содержать значение, которое было отправлено в заголовке Origin запроса.
Если Access-Control-Allow-Origin нет, то браузер считает, что разрешение не получено, и завершает запрос с ошибкой.

Предварительно проверяемые запросы

В кросс-доменном XMLHttpRequest можно указать не только GET/POST, но и любой другой метод, например, PUT, DELETE.

Любое из условий ведёт к тому, что браузер сделает два HTTP-запроса:

- Если метод не **GET**/**POST**/**HEAD**;
- Если заголовок **Content-Type** имеет значение отличное от **application/x-www-form-urlencoded**, **multipart/form-data** или **text/plain**, например application/xml;
- Если устанавливаются другие HTTP-заголовки, кроме **Accept**, **Accept-Language**, **Content-Language**.

Предварительный запрос использует метод **OPTIONS**. Он не содержит тела и содержит название желаемого метода (и при необходимости особые заголовки) в заголовке Access-Control-Request-Method.

На этот запрос сервер должен ответить статусом 200 без тела ответа, указав заголовки:

- Access-Control-Allow-Method : метод;
- Access-Control-Allow-Headers : разрешённые заголовки (при необходимости).

Рассмотрим пример предварительно проверяемого запроса, в котором мы в своём веб-приложении по учёту финансов добавляем нового пользователя:

```
var invocation = new XMLHttpRequest();
var url = 'http://www.financeManager.ru/post-here/';
var body = '<?xml version="1.0"?><person><name>Василий</name></person>';

function addUser(){
  if (invocation){
     invocation.open('POST', url, true);
     invocation.setRequestHeader('X-PINGOTHER', 'pingpong');
     invocation.setRequestHeader('Content-Type', 'application/xml');
     invocation.onreadystatechange = handler;
     invocation.send(body);
  }
}
```

Строка 3 создает тело с XML для отправки запросом POST в строке 8. В строке 9 устанавливается нестандартный заголовок HTTP-запроса **(X PINGOTHER: pingpong**). Такие заголовки не являются частью протокола **HTTP/1.1** , но они могут быть нужны для веб-приложений.
Поскольку в запросе используется тип контента **application/xml** и установлен настраиваемый заголовок, этот запрос предварительно проверяется.

Рассмотрим запрос OPTIONS

```
OPTIONS /post-here/ HTTP/1.1
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type ...
```

- Заголовок **Access-Control-Request-Method** как часть предварительного запроса уведомляет сервер о том, что при отправке основного запроса он будет отправлен методом POST;

- Заголовок **Access-Control-Request-Headers** уведомляет сервер о том, что основной запрос будет отправлен с пользовательскими заголовками **X-PINGOTHER** и **Content-Type**;
- Теперь у сервера есть возможность определить, может ли он принять такой запрос или нет.

Рассмотрим ответ на запрос OPTIONS

```
HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: http://www.financeManager.ru
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
```

- Сервер посылает Access-Control-Allow-Methods и говорит, что POST и GET являются допустимыми методами для запроса соответствующего ресурса.
- Сервер также отправляет Access-Control-Allow-Headers со значением X-PINGOTHER, Content-Type, подтверждая, что это разрешённые заголовки, которые будут использоваться с основным запросом.

После предварительного запроса основной запрос выполняется в обычном режиме.

Чтение нестандартных заголовков ответа

По умолчанию скрипт может прочитать из ответа только простые заголовки, такие как:

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

Например, Content-Type можно получить всегда, а доступ к нестандартным заголовкам нужно открывать явно.

Чтобы JavaScript мог прочитать HTTP-заголовок ответа, сервер должен указать его имя в **Access-Control-Expose-Headers**.
Например,

```
Access-Control-Expose-Headers: X-My-Custom-Header, X-Another-Custom-Header
```

позволяет заголовкам X-My-Custom-Header и X-Another-Custom Header быть прочитанными браузером.

Запрос с куки и авторизующими заголовками

Одна из возможностей XMLHttpRequest — делать проверенные запросы, которые осведомлены о файлах cookie и информации об HTTP-аутентификации. По умолчанию в кросс-доменных вызовах XMLHttpRequest браузеры не отправляют учётные данные. Для этого должен быть установлен специальный флаг для объекта **XMLHttpRequest**.

В примере наше веб-приложение по учёту финансов http://www.financeManager.ru выполняет простой GET-запрос к ресурсу, который устанавливает файлы cookie. Наше веб-приложение может содержать следующий JavaScript:

```
const invocation = new XMLHttpRequest();
const url = 'http://www.example.ru/credentialed-content/';

const getDataWithCredentials => () = {
  if (invocation) {
    invocation.open('GET', url, true);
    invocation.withCredentials = true;
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```

В строке 7 проставляется флаг withCredentials для **XMLHttpRequest**, который нужен для того, чтобы сделать вызов с помощью **Cookies**. Это простой запрос GET, и он не является предварительно проверяемым, но браузер отклонит любой ответ, который не имеет заголовка **Access-Control-Allow-Credentials: true**, и не сделает ответ доступным для вызывающего сайта.
Таким образом, при запросе с **withCredentials** сервер должен вернуть уже не один (как в случае с простым запросом), а два заголовка:
Access-Control-Allow-Origin: домен
Access-Control-Allow-Credentials: true

Пример заголовков ответа сервера для нашего случая:
HTTP/1.1 200 OK
Content-Type:text/html; charset=UTF-8
Access-Control-Allow-Origin: http://financeManager.ru
Access-Control-Allow-Credentials: true

Использование звёздочки ***** в **Access-Control-Allow-Origin** при этом запрещено. Если этого заголовка не будет, то браузер не даст JavaScript доступ к ответу сервера.
Более подробно о CORS можно прочитать по ссылке в дополнительных материалах.

Итоги

- Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность веб-приложению, работающему на одном домене, получить доступ к выбранным ресурсам с сервера на другом домене;
- Выделяют простые и предварительно проверяемые запросы (с предварительным запросом OPTIONS);
- Если сервер отвечает Access-Control-Allow-Origin: * , это означает, что к ресурсу может обращаться любой домен в межсайтовом режиме. Также разрешение может быть только у конкретного домена:
  Access-Control-Allow-Origin: домен ;
- Флаг withCredentials для XMLHttpRequest нужен для того, чтобы сделать вызов с помощью Cookies (используется при HTTP-аутентификации).

Итоги по теме

- localStorage — это специальный объект BOM, с помощью которого можно долговременно хранить информацию на клиенте. Для доступа к данным предоставляется удобный интерфейс для чтения и записи пар ключ-значение (строки);
- Куки — это пары строк ключ-значение, которые могут сохраняться даже после закрытия браузера. Часто используются для авторизации пользователя;
- CORS — это механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность веб-приложению, работающему на одном домене, получить доступ к выбранным ресурсам с сервера на другом домене.

Материалы, использованные при подготовке:

- Cross-Origin Resource Sharing (CORS)
- XMLHttpRequest.withCredentials
- Window.localStorage
- Window.sessionStorage
- Using HTTP cookies