HTML5 API

LoftSchool
от мыслителя к создателю

Содержание

1

HTML API

Это еще один вид API, расширяющий BOM(Browser Object Model)

В современном мире, браузер уже давно перестал быть обычным средством просмотра страниц.

Помимо серфинга страниц, современные браузеры имеют гораздо больший арсенал возможностей: определению местонахождения пользователя, встроенные средства для организации конференц-связи, работа с виртуальной файловой системой, работа со встроенной РБД (реляционная база данных), прямой доступ к видео и звуковой карте вашей системы, и еще огромное количество различных других улучшений, которые делают из вашего браузера многофункциональный программный комплекс.

И естественно, что вы можете использовать эти возможности в своих JS-скриптах.

Ниже перечислены самые популярные составляющие **HTML5 API**.

2

History states

Это механизм, позволяющий создавать цепочку состояний страницы и затем перемещаться по этой цепочке в обоих направлениях при помощи кнопок навигации вперед и назад, восстанавливая данные состояния.

Отличительная особенность заключается в том, что никакой перезагрузки страницы, при использовании кнопок навигации, не происходит. Данный механизм широко используется в **SPA(Single Page Application)**.

Вот так просто можно попросить браузер добавить новое состояние в цепочку.

```
history.pushState({
   prop1: 'value1',
   prop2: 'value2',
}, 'state name', '/some/url');
```

Браузер создаст новое состояние с именем **state name**, присвоит этому состоянию адрес **/some/url** и запишет в него переданный объект.

Собрав таким образом цепочку состояний, мы можем перемещаться по ней при помощи кнопок навигации.

При этом, мы сможем наблюдать, как в адресной строке меняется URL на указанный при добавлении текущего состояния, но перезагрузки страницы при этом не происходит. Таким образом, мы эмитируем переход по страницам, но на самом деле происходит переход по запомненным ранее состояниям.

А вот так можно отловить тот момент, когда происходит смена состояния:

```
window.addEventListener('popstate', function(e) {
   console.log(e.state);
});
```

И именно в этом обработчике мы будем принимать решение, что сделать с данными, сохраненными в состоянии. Например, мы можем восстановить содержимое текстовых полей сохраненными значениями.

3

Local/session storage

Это механизмы, позволяющие хранить данные внутри браузера.

Сразу может возникнуть вопрос: “А зачем? Ведь есть cookies!”

Всё очень просто:

Cookies текущей страницы автоматически отправляются при каждом запросе на сервер. Данные из **local storage** никуда не отправляются.

Cookeis имеют ограничение на размер хранимых данных в несколько килобайт. **Local storage** имеет гораздо большую вместимость - несколько мегабайт (конкретное число зависит от браузера).

Cookies имеют ограниченное время жизни. Данные в **local storage** не имеют ограничения во времени жизни.

Плюс ко всему, отличается и сами механизм работы с ним из JS.
Local storage - это сего лишь объект. И вот как можно с ним работать:

```
localStorage.someData = 'привет!';

console.log(localStorage.someData); //выведет "привет"
```

Гораздо проще, чем с cookies, не правда ли?

Данные в **local storage** не потеряются, даже если вы закроете браузер.

В **local storage** можно хранить только строки.

Таким образом, если вы хотите хранить объект или массив, то необходимо превратить их в строку, а при чтении из local storage - превратить обратно:

```
var data = {
   prop1: 'привет',
   prop2: 'loftschool'
};

localStorage.someData = JSON.stringify(data);

data = JSON.parse(localStorage.someData);
console.log(data); //выведет в консоль объект
```

Для **session storage** применимо всё то, что мы описали для **local storage** за тем лишь исключением, что данные в **session storage** сотрутся сразу после закрытия браузера.

4

FileAPI

Механизм, позволяющий получить содержимое выбранных пользователем файлов в режиме реального времени, что дает возможность, например, посмотреть загружаемый файл в браузере еще до его загрузки на сервер. Так же, есть возможность разделить загружаемый файл на части, прямо в браузере и загрузить его на сервер по частям. Всё это делается при помощи специального объекта - **FileReader**.

Именно через **FileReader** мы можем преобразовать файл, указанный пользователем в набор байт или текст и отобразить в браузере.

Тема **FileAPI** довольно обширна и на этот счет в интернете есть замечательная статья с примерам - **http://www.html5rocks.com/ru/tutorials/file/dndfiles/**

5

WebSocket

Механизм, позволяющий поддерживать с сервером постоянное соединение.

Как известно, протокол http устроен таким образом, что передав запрошенное содержимое, сервер разрывает соединение с браузером.

Но с приходом **WebSocket**, стало возможным поддерживать постоянное соединение браузера с сервером, даже после окончания передачи данных от сервера к барузеру.

Таким образом, создание чата, работающего без перезагрузки страницы - больше не является чем-то фантастическим.

Для начала, необходимо убедиться, что сервер, к которому мы собираемся подключиться, поддерживает **WebSocket**.

Если поддерживает, то необходимо создать **socket**:

```
var socket = new WebSocket("ws://localhost:8080");
```

Как видите, мы подключаемся по протоколу **ws:// к серверу с именем localhost и портом 8080**.

После этого, созданному сокету необходимо добавить обработчик события **message** чтобы иметь возможность получать сообщения от сервера:

```
socket.onmessage = function(event) {
  console.log(event.data);
}
```

Отправлять сообщения на сервер можно при помощи метода **send()**:

```
socket.send('привет!');
```

6

WebWorker

Это механизм, позволяющий запускать какой-либо код параллельно с основным кодом.

Это может быть полезно в тех случаях, когда необходимо реализовать обработку каких-либо больших массивов данных, а нагружать основной поток кода не хочется.

Работать с воркером очень просто. Для начала, его необходимо создать:

```
var worker = new Worker('worker.js');
```

При создании воркера, необходимо указать скрипт, в котором содержится исходный код самого воркера. Это самый обычный JS-скрипт, с обработчиком глобального события **message**:

```
//это код внутри воркера
self.addEventListener('message', function(e) {
   //какие-то операции
});
```

Это нужно только в том случае, если необходимо, чтобы воркер принимал сообщения от основного потока.

Основной поток, в свою очередь, тоже может принимать сообщения от воркера, для этого, в основном коде, необходимо добавить обработчик события **message** воркера:

```
//это код внутри основного файла
worker.addEventListener('message', function(e) {
   console.log(event.data);
};
```

Всё, что умеет воркер - это обмениваться сообщениями с основным кодом и производить затратные вычисления:

```
//это код внутри воркера
self.postMessage('привет!');

//это код внутри основного файла
worker.postMessage('привет!');
```

Заметьте, что внутри вокрера, у вас нет доступа к dom-дереву! Воркер не предназначен для работы с DOM.

7

Geolocation

Механизм, позволяющий получить координаты своего местонахождения.

Несмотря на кажущуюся сложность, использовать этот механизм очень просто:

```
navigator.geolocation.getCurrentPosition(function(geo) {
   console.log(geo);
});
```

Вот и всё! Вызываем метод **getCurrentPosition()** и передаем ему **call-back-функцию**, которая будет вызвана сразу после того, когда браузера определит наше местонахождения. В функцию автоматически будет передана информация о нашем местонахождении.

Так же есть возможность отслеживать перемещение владельца браузера:

```
var
   watchId = navigator.geolocation.watchPosition(function(geo)
{
   console.log(geo);
});
```

Смысл тот же самый, за тем лишь исключением, что **callback**, переданный в **watchPosition**, будет вызван не один раз, а будет вызываться бесконечно, с определенной периодичностью.

watchPosition возвращает идентификатор, который может быть использован для отмены отслеживания:

```
navigator.geolocation.clearWatch(watchId);
```

Более объемные примеры указанных здесь технологий, будут рассмотрены на вебинаре. Не пропустите!