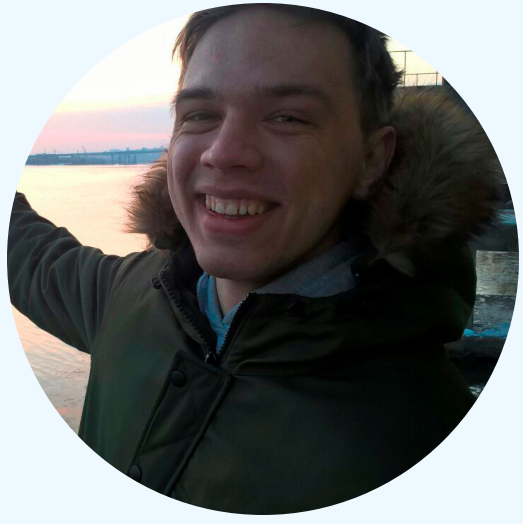
нетология
университет интернет-профессий

ВЕБСОКЕТЫ

БОРИС СТРЕЛЬНИКОВ / ТИНЬКОФФ



БОРИС СТРЕЛЬНИКОВ

Javascript developer Тинькофф

risentveber.ru/ risentveber@gmail.com @javascriptjobs

ВЕБСОКЕТЫ

СОЗДАЕМ ЧАТ

У нас стоит задача создать самый простой чат, который будет позволять в реальном времени отправлять и получать сообщения. Кроме работы с сообщениями чат должен своевременно реагировать на ошибки и сообщать пользователю о них.

XMLHttpRequest

Для начала рассмотрим пример реализации такого чата с помощью уже известного XMLHttpRequest.

Начнем с реализации отправки сообщений. Как это делается с `XMLHttpRequest` вы уже знаете:

```
const url = '/chat';
const xhr = new XMLHttpRequest();
xhr.open('POST', url);
const formData = new FormData();
formData.append('name', 'Дмитрий');
formData.append('message', 'Привет');
xhr.send(formData);
```

ПОЛУЧАЕМ СООБЩЕНИЯ

Далее необходимо реализовать получение сообщений:

```
xhr.open('GET', url);
xhr.send();
xhr.addEventListener('load', () => {
  if (xhr.status === 200) {
    console.log(xhr.responseText);
      // Вывод сообщения с сервера
  }
};
```

ОБРАБАТЫВАЕМ ОШИБКИ

Но помимо сообщений могут возникать ошибки, которые также необходимо обработать:

```
xhr.addEventListener('load', () => {
  if (xhr.status !== 200) {
    console.log(`Произошла ошибка ${xhr.status},
      ${xhr.statusText}`);
  }
  else {
    console.log(xhr.responseText);
  }
});
```

ЧАТ «ПО ТРЕБОВАНИЮ»

Кажется, весь нужный функционал есть — сообщения можно отправлять, сообщения можно получать, ошибки обрабатываются. Но когда это происходит?

Все это происходит только при вызове методов получения и отправки сообщений. То есть в реальном времени сообщения приходить не будут и о новых сообщениях и ошибках мы узнаем только тогда, когда запросим данные с сервера. Получается, чтобы получить новое сообщение нам нужно вызвать метод для получения данных.

Для этого придется либо добавить кнопку «Обновить чат» и постоянно нажимать ее либо повесить таймер, который будет обновлять сообщения каждый определенный промежуток времени. При этом каждый раз соединение будет открываться по-новой, отправляя и получая все дополнительную служебную информацию. А если работать приходится с разными доменнами еще и придется решать проблему CORS.

ПРОБЛЕМЫ XMLHTTPREQUEST-ЧАТА

То есть, получается, в текущей реализации чата с XMLHttpRequest возникают следующие проблемы:

— Нет статуса в реальном времени, нужно обновлять страницу, чтобы получить новое состояние;

— Нет обработки ошибок в реальном времени — про ошибку мы узнаем только при отправке сообщения или при обновлении страницы;

— Соединение каждый раз открывается по-новой;

— Вместе с сообщением каждый раз приходится отправлять много дополнительной информации.

ПОТЕНЦИАЛЬНОЕ РЕШЕНИЕ

Как можно решить проблемы?

- Хотелось бы такое решение, которое откроет соединение и будет держать его открытым до тех пор, пока мы его не закроем сами;
- Чтобы в этом решении вся информация приходила своевременно и пользователь узнавал о новых сообщениях сразу без повторного запроса данных;
- Так как соединение постоянно открыто и нет необходимости каждый раз повторно отправлять служебную информацию, то неплохо бы, чтобы отправлялась и получалась только нужная информация.

«

Websocket *– протокол полнодуплексной связи (может одновременно передавать и принимать) поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени. При использовании вебсокетов снимаются ограничения на передачу данных и это позволяет пересылать любый данные на любой домен безопасно и почти без лишнего сетевого трафика.*

ОТКРЫТИЕ СОЕДИНЕНИЯ

Для того, чтобы начать работать с вебсокетами нужно открыть вебсокет-соединение. Сделать это достаточно просто — нужно только создать объект `Websocket`, указав в качестве параметра специальный URL с протоколом `ws://`:

```
const connection = new WebSocket('ws://example.com/');
```

Но само по себе открытие соединения ничего не даст, нужно также сообщить соединению, как будут обрабатываться сообщения и ошибки, что должно происходить при открытии и закрытии соединения.

СОБЫТИЯ СОЕДИНЕНИЯ

У объекта `connection` есть 4 события, на которые мы можем подписаться с помощью `addEventListener`. Рассмотрим их по порядку.

open

Событие `open` срабатывает при открытии соединения. В качестве примера просто выведем в консоль информацию о том, что соединение открыто. На практике же здесь может быть более сложная логика, которую необходимо выполнить при открытии соединения:

```
connection.addEventListener('open', () => {
  console.log('Вебсокет-соединение открыто');
});
```

Событие `open` срабатывает только один раз при открытии соединения и в нем обычно размещают различную логику инициализации. Например, если мы начали попытку соединения и показали индикатор соединения, то при соединение здесь мы его можем скрыть.

СОБЫТИЕ `message`

Событие `message` происходит при получении сообщения. Данные приходят в событии, поэтому необходимо передать его в качестве аргумента. В объекте `event` много разной информации, но нас интересует только сообщение, которое пришло с сервера, оно хранится в свойстве `data`:

```
connection.addEventListener('message', event => {
  console.log(`Получено сообщение: ${event.data}`);
});
```

СОБЫТИЕ `close`

Событие `close`, как можно догадаться, срабатывает при закрытии соединения:

```
connection.addEventListener('close', event => {
  console.log('Вебсокет-соединение закрыто');
});
```

ПРИЧИНА ЗАКРЫТИЯ СОЕДИНЕНИЯ

В событии `close`, которое приходит при закрытии соединения можно также найти полезную информацию, например, код и причину закрытия. Например, код **1000** означает нормальное закрытие, а код **1012** означает, что событие закрыто по причине рестарта сервера. Кодов закрытия на самом деле немало и, в зависимости от него, мы можем предпринимать различные действия на клиентской части.

СОБЫТИЕ `error`

И существует еще одно событие `error`, которое случается, если произошла ошибка:

```
connection.addEventListener('error', error => {
  console.log(`Произошла ошибка: ${error.data}`);
});
```

В зависимости от реализации в событии ошибки придет информация об ошибке.

ПОД КАПОТОМ ВЕБСОКЕТОВ

Протокол WebSocket работает над протоколом HTTP и все начинается, как при работе с обычным HTTP. При соединении браузер отправляет специальные заголовки, спрашивая: «поддерживает ли сервер вебсокеты?».

Выглядит это так:

```
GET /demo HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: example.com
Origin: http://example.com
```

ОТВЕТ СЕРВЕРА

Если сервер поддерживает вебсокеты, то он присылает в ответ заголовок вида:

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://example.com
WebSocket-Location: ws://example.com/demo
```

Если браузер это устраивает, то TCP-соединение остается открытым и дальше начинается общение через вебсокет.

ОТПРАВЛЯЕМ СООБЩЕНИЯ

Для отправки сообщений используется метод `send`, в котором можно пересылать любые данные.

Сообщение в виде простой строки отправляется очень просто, необходимо передать отправляемую строку в качестве аргумента в метод `send`:

```
connection.send('Простое сообщение, отправленное
  через websocket');
```

ПЕРЕСЫЛАЕМ ОБЪЕКТ

Как было сказано ранее, мы можем переслать любое сообщение. Например, помимо простой строки мы можем переслать объект:

```
connection.send({
  author: 'Дмитрий',
  message: 'Простое сообщение в объекте,
    отправленном через websocket'
});
```

КОНВЕРТИРУЕМ ОБЪЕКТ

Но, если отправить объект таким образом, то, поскольку в `send` можно отправлять только строку, то и объект преобразуется в строку и в результате вместо объекта отправится `"[object Object]"`. Это не совсем то, что нам нужно, поэтому отправляемый **объект необходимо сконвертировать в JSON-строку**. Сделать это можно с помощью `JSON.stringify` :

```
connection.send(JSON.stringify({
  author: 'Дмитрий',
  message: 'Простое сообщение в объекте,
    отправленном через websocket'
}));
```

Теперь на сервер отправится тот объект, который нам нужно отправить и на сервере он может быть обработан.

ПОЛУЧАЧЕМ СООБЩЕНИЯ

Для получения сообщений используется упомянутый выше колбек `onmessage`. В случае со строкой, как было сказано выше все просто — читаем сообщение из `event.data`:

```
1 connection.addEventListener('message', event => {
2   console.log(event.data);
3 });
```

ПОЛУЧАЕМ ОБЪЕКТ

Но часто с сервера приходит не просто строка, а более сложный объект. В этом случае он тоже придет в виде строки, которую нужно распарсить. Сделать это можно с помощью `JSON.parse`:

```
connection.addEventListener('message', event => {
  var message = JSON.parse(event.data);
  console.log('Получено сообщение:');
  console.log(message);
};
```

ЗАКРЫВАЕМ СОЕДИНЕНИЕ

Когда мы заканчиваем работать с вебсокет-соединением, нам нужно его закрыть. Сделать это мы можем с помощью метода `close`:

```
connection.close();
```

Если соединение еще не было закрыто и еще не был запущен процесс его закрытия, то вызов этого метода инициализирует процесс закрытия.

КОД ЗАКРЫТИЯ

Кроме того, в методе `close` мы можем передать код закрытия и сообщение, как, например, ниже:

```
connection.close(1000, 'Работа закончена');
```

После того, как соединение будет закрыто произойдет событие `close` и событие `message` больше не будет вызываться до тех пор, пока мы снова не откроем соединение.

ЗАКРЫВАЕМ СОЕДИНЕНИЕ И СТРАНИЦУ

Часто нужно закрыть соединение при уходе со страницы. Сделать это можно добавив закрытие к событию `beforeunload`:

```
window.addEventListener('beforeunload', () => {
  connection.onclose = function () {};
  connection.close()
};
```

В этом случае, при закрытии страницы соединение закроется.

ПОДДЕРЖКА БРАУЗЕРАМИ

На текущий момент вебсокеты работают во всех современных браузерах IE10+, Edge, FF11+, Chrome 16+, Safari 6+, Opera 12.5+. В более старых версиях FF, Chrome, Safari, Opera есть поддержка черновых редакций протокола.

ПОЛЬЗА ВЕБСОКЕТОВ

вебсокеты будут полезны, если вы создаете:

- веб-приложения, с интенсивным обменом данными, требовательные к скорости обмена и каналу;
- приложения, следующие стандартам;
- «долгоиграющие» веб-приложения;
- комплексные приложения с множеством различных асинхронных блоков на странице;
- кросс-доменные приложения.

ПОЛЕЗНЫЕ ССЫЛКИ

- MDN. WebSocket
- JavaScript.ru. WebSocket
- TutorialsPoint. WebSocket
- Introducing WebSockets: Bringing Sockets to the Web
- An Introduction to WebSockets
- Writing WebSocket client applications
- Echo Test

нетология
университет интернет-профессий

Задавайте вопросы и напишите отзыв о лекции!

БОРИС СТРЕЛЬНИКОВ

 risentveber.ru/

 risentveber@gmail.com

 @javascriptjobs