Лекция 17

NodeJS и Websocksets

# Содержание

[Содержание 2](#_Toc520193023)

[Способы обмена данных между сервером и клиентом. 3](#_Toc520193024)

[WebSockets 4](#_Toc520193025)

[Реализация WebServices во фронтенд части 6](#_Toc520193026)

[NodeJS 8](#_Toc520193027)

[Варианты применения в которых NodeJS отлично себя показывает 8](#_Toc520193028)

[Варианты применения в которых NodeJS не очень хорош 9](#_Toc520193029)

[Простейший сервер на NodeJS 9](#_Toc520193030)

[Использование WebSocksets с NodeJS 10](#_Toc520193031)

[Express JS 11](#_Toc520193032)

[Socket.IO 12](#_Toc520193033)

[Дополнительные материалы 14](#_Toc520193034)

[Задание для закрепления материала 15](#_Toc520193035)

# Способы обмена данных между сервером и клиентом.

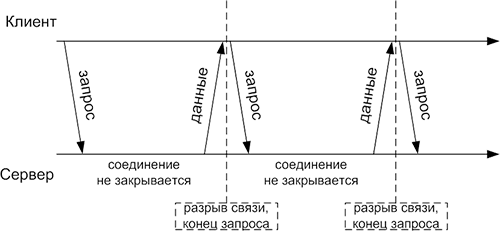
Способов обмена множество, но в данном контексте остановимся на способах, используемых в веб разработке. Клиентом здесь всегда выступает браузер, а он работает по протоколу HTTP(S) Какой самый простой метод обменяться данным ис сервером? Открыть соединение методом GET или POST, послать какие-то данные, получить ответ, обработать. Это обычный простейший подход, в браузере для этого есть обьект XMLHttpRequest над которым существует бесчисленное множество оберток. Один из вариантов, который очень хорош во многих случаях – AJAX (Asynchronous Javascript And Xml) , он позволяет выполнять массу задач, таких как подгрузка элементов интерфейса, динамических данных с сервера, организация живого поиска, запрос прогресса загрузки большого файла с сервера или статуса работы на сервере какой то задачи. При использовании легковесного бэкэнда (например связок nginx + различные fastCGI, nodejs) с помощью Ajax можно имитировать практически непрерывное соединение. Бэкэнд может достаточно точно определять клиента онлайн и достаточно быстро менять содержимое на клиентах. Исходя из этого, такой способ обмена данными между браузером и сервером (обычный HTTP запрос) предпочтителен в силу его простоты.

Но, у такого подхода инициируемого браузером, есть свои недостатки, некоторые из них могут быть критичными, например:

* Большое количество клиентов периодически запрашивающих данные порождают большой трафик. Стоит упомянуть, что в обычном HTTP запросе при каждом запросе на сервер посылаются и получаются заголовки, обьем которых может быть велик. Одних только куки может быть немало, и часто может быть так, что обьем передаваемых данных куда меньше заголовков.
* Нагруженный/медленный бэкэнд, имеющий свои высокие накладные расходы, например на на открытие соединений с БД, запуск процессов.
* Невозможность инициировать запрос со стороны сервера
* Отсутствие данных в реальном времени, всегда с задержкой.

Помимо AJAX существует другой подход, в общем называемый COMET, который подразумевает постоянное соединение с сервером, в результате чего вебсервер получает возможность отправлять данные клиентам. На практике большинство проблем решаются с помощью простейшей реализации COMET- это long-polling запросы.

Вот как это выглядит:



Браузер отправляет запрос на сервер и не разрывает его до тех пор, пока не получит ответ. Этот метод очень хорошо подходит в случае когда долго приходится ждать ответа от сервера, и количество сообщений небольшое.

Для реализации же полноценного постоянного соединения все современные браузеры имеют поддержку WebSockets

# WebSockets

WebSocket – это протокол, предназначенный для снятия ограничений обмена данных между браузером и сервером. Он активирует двустороннее общение между сервером и клиентом, для обсепечения безопасности использует origin-based модель браузера. Протокол содержит обмен рукопожатиями (так называемый handshake) и обмен пакетами сообщений на основе TCP протокола. Это не надстройка над HTTP, это совершенно другой протокол.   
Цель разработки этого протокола – предоставить механизм для веб-приложений, нуждающихся в двустороннем общений без открытия множественных HTTP соединений, таких, какие порождаются с XMLHttpRequest, iframe, long-polling.

Для открытия соединения достаточно создать обьект WebSocket, указав в нем специальный протокол ws

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

У объекта socket есть четыре коллбэка: один при получении данных и три – при изменениях в состоянии соединения:

socket.onopen = function() {

alert("Connection established.");

};

socket.onclose = function(event) {

if (event.wasClean) {

alert('Connection was closed');

} else {

alert('Connection was reset'); // for example, server process killed

}

alert('Code: ' + event.code + ' Reason: ' + event.reason);

};

socket.onmessage = function(event) {

alert("Data received " + event.data);

};

socket.onerror = function(error) {

alert("Error " + error.message);

};

Для посылки данных используется метод send –

socket.send('Hi');

Для успешного обмена сервер должен поддерживать WebSocket. Давайте посмотрим каким образом устанавливается соединение между браузером и сервером. Браузер отправляет такие заголовки:

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Origin: http://localhost

Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==

Sec-WebSocket-Version: 13

Описания заголовков:

**GET, Host**

Стандартные HTTP-заголовки из URL запроса

**Upgrade, Connection**

Указывают, что браузер хочет перейти на websocket.

**Origin**

Протокол, домен и порт, откуда отправлен запрос.

**Sec-WebSocket-Key**

Случайный ключ, который генерируется браузером: 16 байт в кодировке Base64.

**Sec-WebSocket-Version**

Версия протокола. Текущая версия: 13.

Все заголовки, кроме GET и Host, браузер генерирует сам, без возможности вмешательства JavaScript. Это значит что силами JS сформировать и подделать такой запрос нельзя, указанные заголовки нельзя подменить с помощью метода SetRequestHeader

Сервер анализирует заголовки и принимает решение разрешает ли он подключение с указанного в Origin домена. Если сервер понимает вебсокеты, то он выдаст следующий ответ, говорящий о готовности перехода на протокол WebSocket

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=

Здесь строка **Sec-WebSocket-Accept** представляет собой перекодированный по специальному алгоритму ключ **Sec-WebSocket-Key**. Браузер использует её для проверки, что ответ предназначается именно ему.

Соединение вебсокета можно открывать как по ws:// так и по wss:// протоколу. WSS это шифрованный протокол WebSocket аналогично как HTTP/HTTPS.

Общение по протоколу WebSocket просиходит с помощью обмена несколькими видами пакетов (фреймов). Есть фреймы с данными (data frames) и управляющие фреймы (control frames), которые предназначены дя проверки связи и закрытия соединения.

**PING / PONG**

В протокол встроена проверка связи при помощи управляющих фреймов типа PING и PONG.

Тот, кто хочет проверить соединение, отправляет фрейм PING с произвольным телом. Его получатель должен в разумное время ответить фреймом PONG с тем же телом.

Этот функционал встроен в браузерную реализацию, так что браузер ответит на PING сервера, но управлять им из JavaScript нельзя.

Иначе говоря, сервер всегда знает, жив ли посетитель или у него проблема с сетью.

**Чистое закрытие**

При закрытии соединения сторона, желающая это сделать (обе стороны в WebSocket равноправны) отправляет закрывающий фрейм (опкод 0x8), в теле которого указывает причину закрытия.

В браузерной реализации эта причина будет содержаться в свойстве reason события onclose.

**Коды закрытия**

Коды закрытия вебсокета event.code, чтобы не путать их с HTTP-кодами, состоят из 4 цифр:

**1000 -**Нормальное закрытие.

**1001** -Удалённая сторона «исчезла». Например, процесс сервера убит или браузер перешёл на другую страницу.

**1002** - Удалённая сторона завершила соединение в связи с ошибкой протокола.

**1003** -Удалённая сторона завершила соединение в связи с тем, что она получила данные, которые не может принять. Например, сторона, которая понимает только текстовые данные, может закрыть соединение с таким кодом, если приняла бинарное сообщение.

# Реализация WebServices во фронтенд части

HTML страница с формой:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

</head>

<body>

<!-- форма для отправки сообщений -->

<form name="publish">

<input type="text" name="message"/>

<input type="submit" value="Отправить"/>

</form>

<!-- здесь будут появляться входящие сообщения -->

<div id="subscribe"></div>

<script src="browser.js"></script>

</body>

</html>

JS код выполняемый на странице

if (!window.WebSocket) {

document.body.innerHTML = 'WebSocket is not supported in this browser.';

}

// создать подключение

var socket = new WebSocket("ws://localhost:8081");

// send message from form publish

document.forms.publish.onsubmit = function() {

var outgoingMessage = this.message.value;

socket.send(outgoingMessage);

return false;

};

// incoming messages handler

socket.onmessage = function(event) {

var incomingMessage = event.data;

showMessage(incomingMessage);

};

// show message inside div#subscribe

function showMessage(message) {

var messageElem = document.createElement('div');

messageElem.appendChild(document.createTextNode(message));

document.getElementById('subscribe').appendChild(messageElem);

}

# NodeJS

Реализация COMET на стороны сервера может производиться любым удобным способом. Fullstack JS разработчик скорее всего выберет NodeJS. Это очень мощное серверное решение позволяющее полноценно поднимать HTTP вебсервер и Websocket сервер.

Целью этой лекции не является описание всех возможностей NodeJS, но перечислить области его применения и те области, которых стоит избегать – это полезная информация.

## Варианты применения в которых NodeJS отлично себя показывает

**JSON API**

Node.js блестяще справляется с построением легковесных REST / JSON интерфейсов.

Неблокирующй ввод-вывод и JavaScript делают Node отличным вариантом для написания обертки вокруг базы данных или веб-сервиса, которая общается с клиентом в формате JSON.

**Одностраничные приложения**

Если вы пишете сложное AJAX-приложение вроде Gmail, Node тоже очень хорош. Для современных веб-приложений, делающих большую часть работы в браузере, отлично подходит сервер, который может одновременно обрабатывать тысячи запросов и имеет низкое время отклика. Возможность повторного использования одного и того же кода, например валидации, на сервере и клиенте — тоже плюс.

**Использование инструментов командной строки**

Пока Node.js еще очень молод, есть соблазн написать специально для него всё, что только может понадобиться. Но может оказаться гораздо более выгодным использовать огромный выбор уже существующих инструментов командной строки. Node способен порождать тысячи дочерних процессов и работать с их выходными потоками данных, что может дать серьёзные преимущества тем, кто не хочет изобретать велосипеды.

**Потоковая обработка данных**

Традиционные стеки веб-приложений обычно трактуют запросы и ответы HTTP, как атомарные события. Но на самом деле это потоки, и приложения Node.js могут воспользоваться этим фактом. Отличные примеры — обработка файлов во время загрузки на сервер или передача данных между разными слоями.

**Системы мягкого реального времени**

Ещё одна отличная черта Node.js — легкость, с которой можно создавать системы мягкого реального времени. Мессенджеры, чаты, прием ставок на спортивные события или интерфейсы к IM-протоколам.

Это JavaScript, и время отклика может сильно варьировать, если вдруг вклинится сборщик мусора (который, увы, блокирует поток выполнения JavaScript). Так что не пытайтесь построить на Node систему жесткого реального времени с гарантированным временем отклика.

## Варианты применения в которых NodeJS не очень хорош

**Тяжелые вычисления**

Хоть я и люблю Node.js, но есть несколько случаев, когда его применение лишено смысла. Самый очевидный из них — приложения с длительными сложными вычислениями и небольшим количеством операций ввода-вывода. Так что если вы собираетесь писать перекодировщик видео, искусственный интеллект или ещё какую цифродробилку, пожалуйста, не используйте Node.js. Это в принципе возможно, но лучше писать на C/C++.

С другой стороны, node.js позволяет легко писать дополнения на C++, так что вы, конечно, можете использовать его как скриптовой движок поверх ваших сверхсекретных алгоритмов.

**Простые приложения CRUD/HTML**

Хотя Node.js может со временем стать классным инструментом для любых типов веб-приложений, пока не стоит надеяться, что он даст вам больше преимуществ по сравнению с PHP, Ruby или Python.

Несмотря на то, что сейчас уже появляются неплохие фреймворки для Node, пока ещё и близко нет ничего сравнимого с Rails, Laravel или Django. Если ваше приложение в основном занимается заполнением шаблонов HTML данными из базы, Node.js пока не даст вам каких-либо ощутимых преимуществ.

## Простейший сервер на NodeJS

В node большое количество модулей, некоторые из них позволяют поднимать вебсерверы. Рассмотрим на примере модуля node-static предназначенного для поднятия сервера для раздачи статичных файлов.

npm install node-static

Код сервера

var http = require('http');

var static = require('node-static');

var file = new static.Server('.');

http.createServer(function(req, res) {

file.serve(req, res);

}).listen(8080);

console.log('Server running on port 8080');

# Использование WebSocksets с NodeJS

Рассмотрим прототип чата на WebSocket и Node.JS.

HTML: посетитель отсылает сообщения из формы и принимает в div

<!— simple form -->

<form name="publish">

<input type="text" name="message">

<input type="submit" value="Send">

</form>

<!—new messages will appear here -->

<div id="subscribe"></div>

Код на клиенте:

// open connection

var socket = new WebSocket("ws://localhost:8081");

// send data to publish

document.forms.publish.onsubmit = function() {

var outgoingMessage = this.message.value;

socket.send(outgoingMessage);

return false;

};

// input messages handler

socket.onmessage = function(event) {

var incomingMessage = event.data;

showMessage(incomingMessage);

};

// show message inside div#subscribe

function showMessage(message) {

var messageElem = document.createElement('div');

messageElem.appendChild(document.createTextNode(message));

document.getElementById('subscribe').appendChild(messageElem);

}

Серверный код с использованием модуля ws:

var WebSocketServer = new require('ws');

// подключенные клиенты

var clients = {};

// WebSocket-server on port 8081

var webSocketServer = new WebSocketServer.Server({

port: 8081

});

webSocketServer.on('connection', function(ws) {

var id = Math.random();

clients[id] = ws;

console.log("new connection " + id);

ws.on('message', function(message) {

console.log('received message ' + message);

for (var key in clients) {

clients[key].send(message);

}

});

ws.on('close', function() {

console.log('connection closed ' + id);

delete clients[id];

});

});

# Express JS

Вы возможно уже слышали такую аббревиатуру как MEAN или MERN. Это две популярные связки технологий. Где M – MongoDB, E – ExpressJS, A- AngularJS, R-ReactJS, N – NodeJs

Разработчики стремятся к какой то унификации и упорядочении. NodeJS имеет все необходимые встроенные средства для работы в качестве сервера, но веб фреймворки такие как ExpressJS добавляют дополнительное удобство в использовании.

Давайте посмотрим как выглядит простейший сервер на ExpressJS

var express = require('../../');

var app = express();

app.get('/', function(req, res){

res.send('Hello World');

});

if (!module.parent) {

app.listen(3000);

console.log('Express started on port 3000');

}

Связку MEAN или MERN мы использовать в этой лекции не будем, но попробуем на чистом NodeJS в итоге вы возможно писать и не будете, а вот использовать Node вкупе с популярным фрейморком и возможно самой популярным модулем Socket.IO - это позволит вывести приложение на более высокий уровень абстракции.

# Socket.IO

Это такая библиотека, которая специально создана для организации двустороннего общения браузера и сервера. С помощью нее мы и попробуем переписать наше простое приложение вебчата.

Вебчат на Socket.IO

Код формы помещенный в index.html:

<!doctype html>

<html>

<head>

<title>Socket.IO chat</title>

<style>

\* { margin: 0; padding: 0; box-sizing: border-box; }

body { font: 13px Helvetica, Arial; }

form { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }

form input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }

form button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }

#messages { list-style-type: none; margin: 0; padding: 0; }

#messages li { padding: 5px 10px; }

#messages li:nth-child(odd) { background: #eee; }

#messages { margin-bottom: 40px }

</style>

</head>

<body>

<ul id="messages"></ul>

<form action="">

<input id="m" autocomplete="off" /><button>Send</button>

</form>

<script src="https://cdn.socket.io/socket.io-1.2.0.js"></script>

<script src="https://code.jquery.com/jquery-1.11.1.js"></script>

<script>

$(function () {

var socket = io();

$('form').submit(function(){

socket.emit('chat message', $('#m').val());

$('#m').val('');

return false;

});

socket.on('chat message', function(msg){

$('#messages').append($('<li>').text(msg));

window.scrollTo(0, document.body.scrollHeight);

});

});

</script>

</body>

</html>

Код сервера помещенный в Index.js

var app = require('express')();

var http = require('http').Server(app);

var io = require('socket.io')(http);

var port = process.env.PORT || 3000;

app.get('/', function(req, res){

res.sendFile(\_\_dirname + '/index.html');

});

io.on('connection', function(socket){

console.log('a user connected');

socket.on('chat message', function(msg){

io.emit('chat message', msg);

});

});

http.listen(port, function(){

console.log('listening on \*:' + port);

});

Для запуска сервера перейдите в директорию с файлами на выполните

> node index.js

# Дополнительные материалы

<http://expressjs.com/en/starter/installing.html>

<https://socket.io/get-started/chat/>

<http://learn.javascript.ru/websockets>

<https://tools.ietf.org/html/rfc6455>

<http://learn.javascript.ru/ajax-nodejs>

# Задание для закрепления материала

Нужно добавить к существующему калькулятору одну кнопку. Это должен быть какой-то секретный расчет, логика которого спрятана в глубинах бэкэнда. Включите фантазию. Отправка данных для расчета и результат должны приходить по протоколу WebSocket. Потребуется поднять WebSocket сервер на NodeJS который будет принимать запрос и рассчитывать данные по секретной формуле.