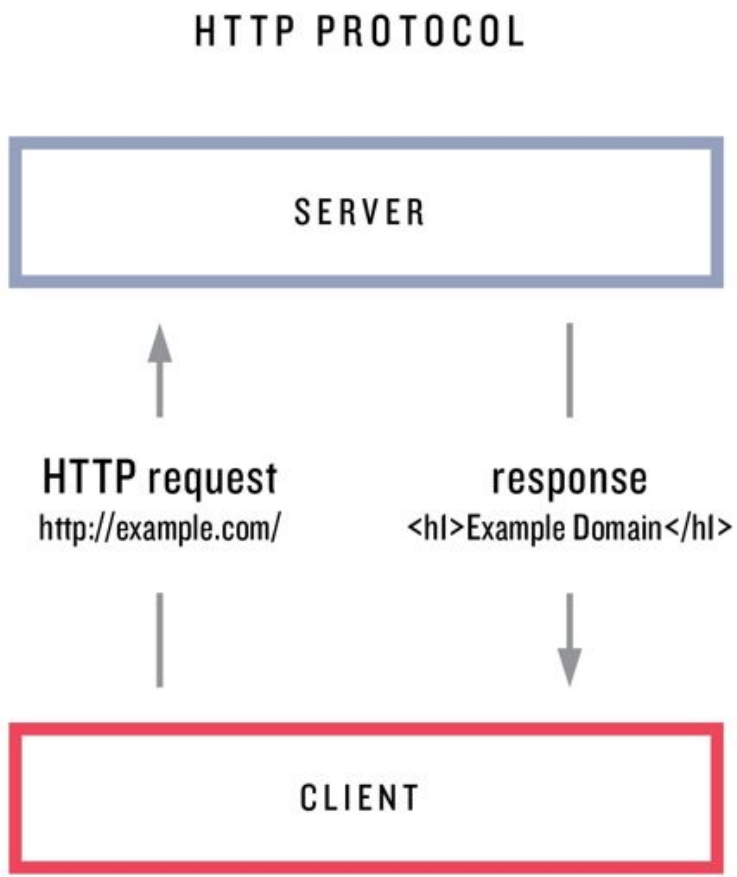
Подписочная модель обмена сообщениями

Вспомним схему взаимодействия HTTP



Постановка задачи

Представим что нам необходимо написать {чат/игру/систему оповещений} для нескольких пользователей. Каким образом мы будет оповещать подключившихся пользователей об изменениях в системе?

Учитываем что сервер ничего не знает о подключениях, т.к. инициирующей стороной является клиент.

Открытый вопрос… что же мы можем сделать?

Periodic polling (Частые опросы)

Самый простой способ получать новую информацию от сервера – периодический опрос. То есть, регулярные запросы на сервер вида: «Привет, я здесь, у вас есть какая-нибудь информация для меня?». Например, раз в 10 секунд.

В ответ сервер, во-первых, помечает у себя, что клиент онлайн, а во-вторых посылает весь пакет сообщений, накопившихся к данному моменту.

Periodic polling (Частые опросы)

Это работает, но есть и недостатки:

- Сообщения передаются с задержкой до 10 секунд (между запросами).
- Даже если сообщений нет, сервер «атакуется» запросами каждые 10 секунд, даже если пользователь переключился куда-нибудь или спит. С точки зрения производительности, это довольно большая нагрузка.

Так что, если речь идёт об очень маленьком сервисе, подход может оказаться жизнеспособным, но в целом он нуждается в улучшении.

Long polling (Длинные запросы)

Длинные опросы – это самый простой способ поддерживать постоянное соединение с сервером, не используя при этом никаких специфических протоколов.

Его очень легко реализовать, и он хорошо подходит для многих задач

Long polling (Длинные запросы)

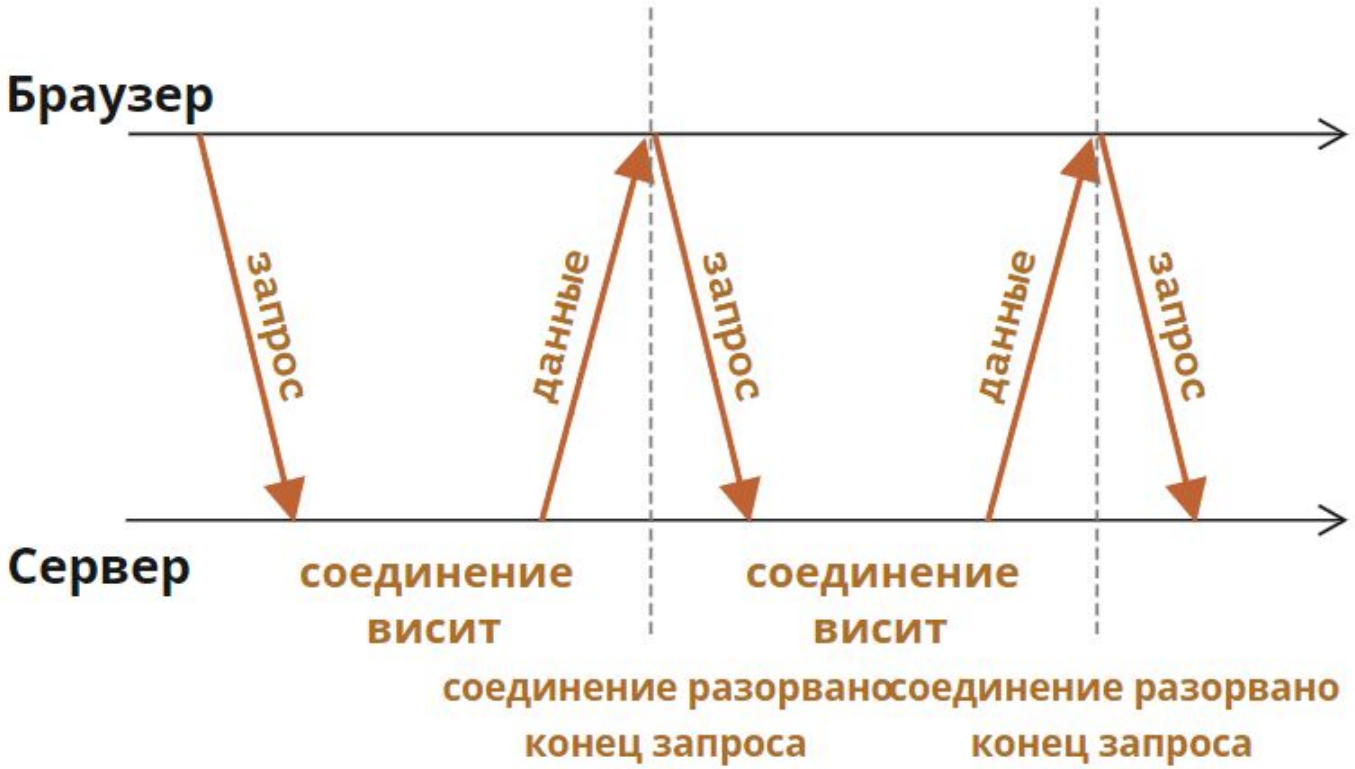
«Длинные опросы» – гораздо лучший способ взаимодействия с сервером.

Они также очень просты в реализации, и сообщения доставляются без задержек.

Как это происходит:

1. Запрос отправляется на сервер.
2. Сервер не закрывает соединение, пока у него не возникнет сообщение для отсылки.
3. Когда появляется сообщение – сервер отвечает на запрос, посылая его.
4. Браузер немедленно делает новый запрос.

Long polling (Длинные запросы)



Long polling (Длинные запросы)

Для данного метода ситуация, когда браузер отправил запрос и удерживает соединение с сервером в ожидании ответа, является стандартной. Соединение прерывается только доставкой сообщений.

Если соединение будет потеряно, скажем, из-за сетевой ошибки, браузер всё ровно будет продолжать посылать новые запросы.

Рассмотрим код клиентской функции subscribe, которая реализует длинные опросы.

```javascript
async function subscribe() {
  let response = await fetch("/subscribe");

  if (response.status == 502) {
    // Статус 502 - это таймаут соединения;
    // возможен, когда соединение ожидало слишком долго
    // и сервер (или промежуточный прокси) закрыл его
    // давайте восстановим связь
    await subscribe();
  } else if (response.status != 200) {
    // Какая-то ошибка, покажем её
    showMessage(response.statusText);
    // Подключимся снова через секунду.
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Получим и покажем сообщение
    let message = await response.text();
    showMessage(message);
    // И снова вызовем subscribe() для получения следующего сообщения
    await subscribe();
  }
}
```

Сервер должен поддерживать много ожидающих соединений.

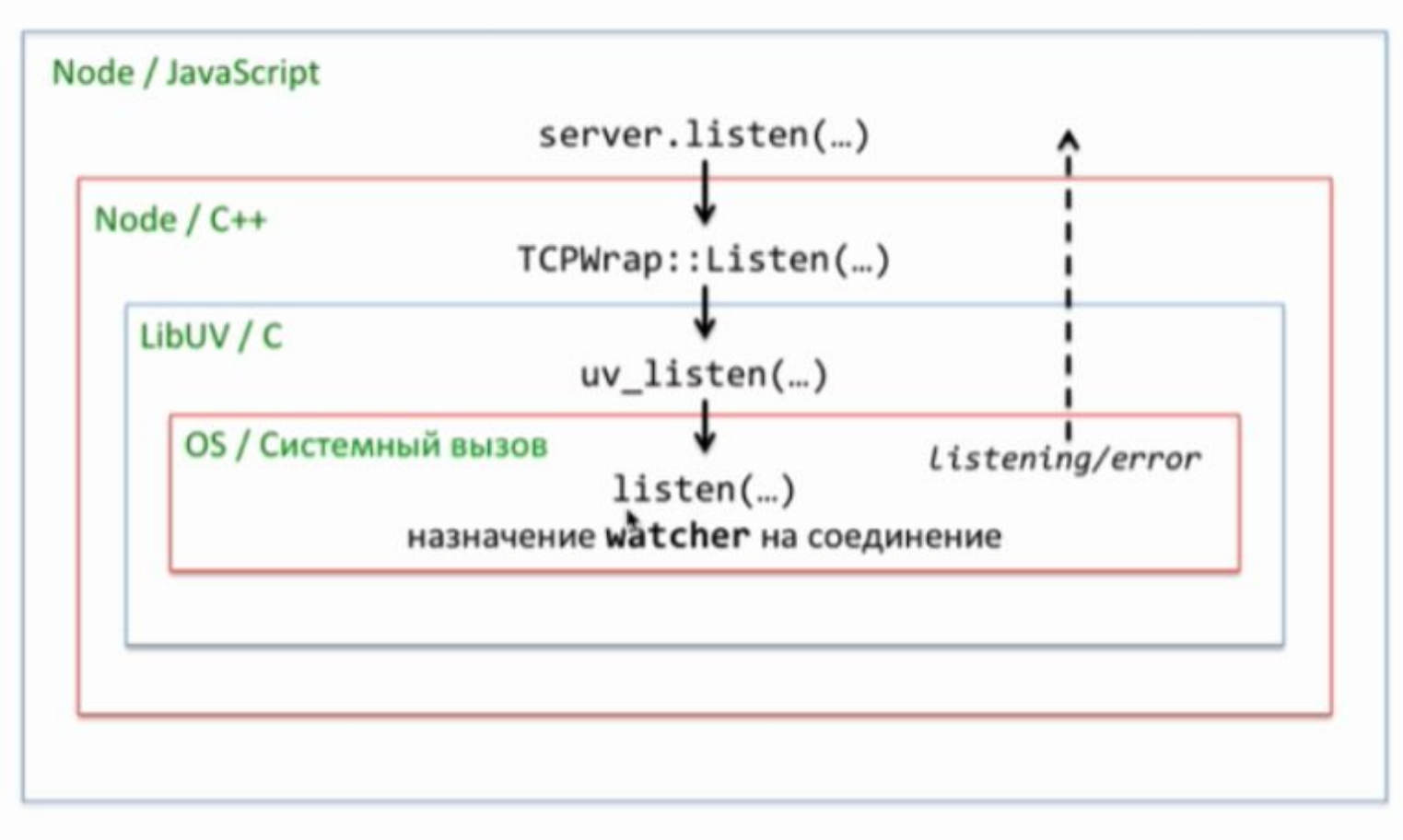
Архитектура сервера должна быть способна работать со многими ожидающими подключениями.

Некоторые серверные архитектуры запускают отдельный процесс для каждого соединения. Для большого количества соединений будет столько же процессов, и каждый процесс занимает значительный объём памяти. Так много соединений просто поглотят всю память.

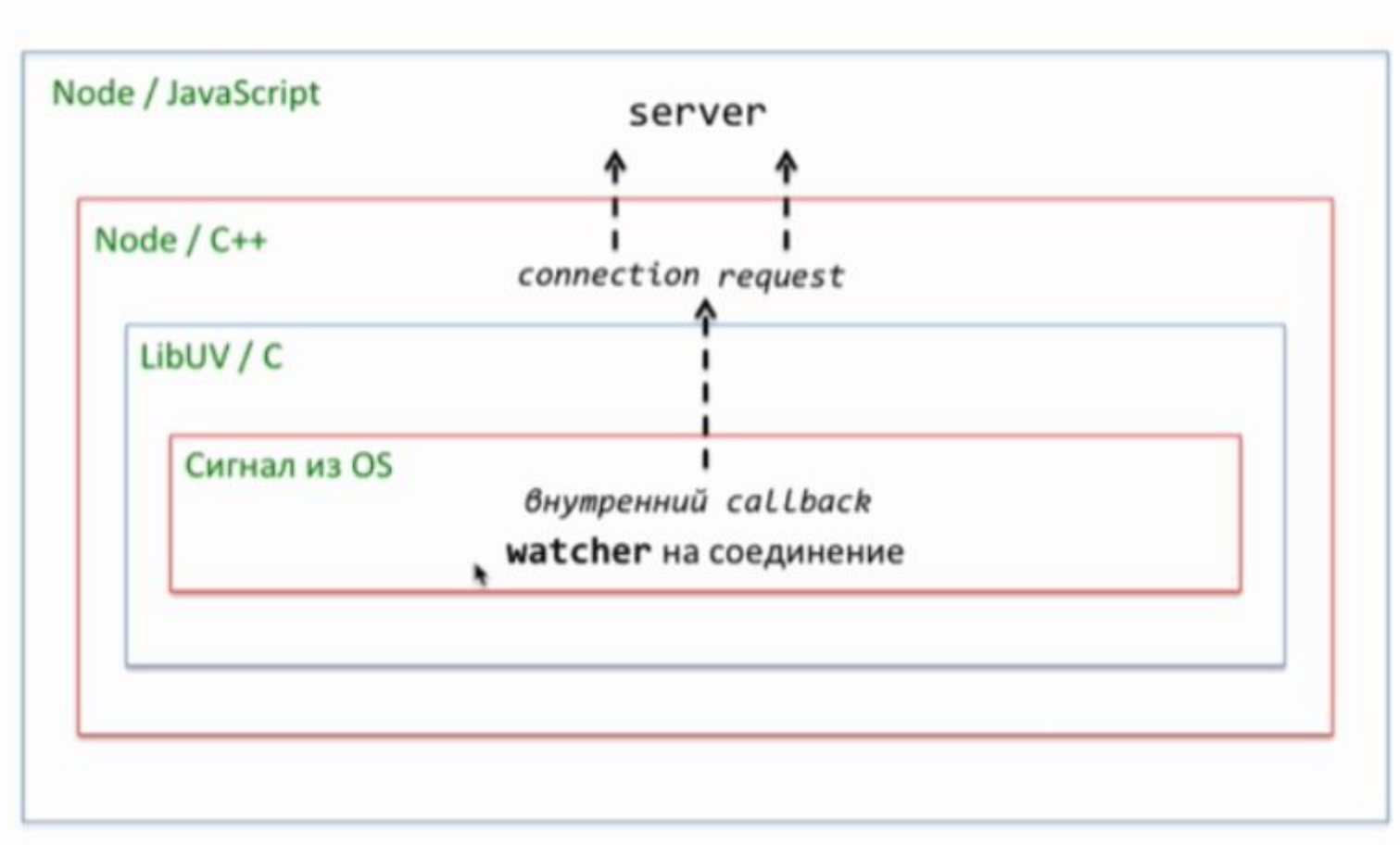
Часто такая проблема возникает с бэкендом, написанными на PHP или Ruby, но технически дело не в языке, а в реализации. На большинстве современных языков можно написать подходящий сервер, но на некоторых это проще сделать.

Бэкенды, написанные с помощью Node.js, обычно не имеют таких проблем.

Почему Node.JS это ОК для Long Pooling’a?



Почему Node.JS это ОК для Long Pooling’а?



Long polling (Длинные запросы)

Длинные опросы прекрасно работают, когда сообщения приходят редко.

Если сообщения приходят очень часто, то схема приёма-отправки сообщений, приведённая выше, становится похожей на «пилу».

Каждое сообщение – это отдельный запрос, с заголовками, авторизацией и так далее.

Поэтому в этом случае предпочтительнее использовать другой метод, такой как, например, WebSocket.

WebSocket

Протокол WebSocket, обеспечивает возможность обмена данными между браузером и сервером через постоянное соединение. Данные передаются по нему в обоих направлениях в виде «пакетов», без разрыва соединения и дополнительных HTTP-запросов.

WebSocket особенно хорош для сервисов, которые нуждаются в постоянном обмене данными, например онлайн игры, торговые площадки, работающие в реальном времени, и т.д.

WebSocket

Чтобы открыть веб-сокет-соединение, нам нужно создать объект new WebSocket, указав в url-адресе специальный протокол ws:

```
let socket = new WebSocket("wss://app.herokuapp.com");
```

Протокол wss://, использующий шифрование.
Это как HTTPS, но только для веб-сокетов.

WebSocket’ы без TLS/SSL = ws://

WebSocket

Как только объект WebSocket создан, мы должны слушать его события. Их всего 4:

- open – соединение установлено,
- message – получены данные,
- error – ошибка,
- close – соединение закрыто.

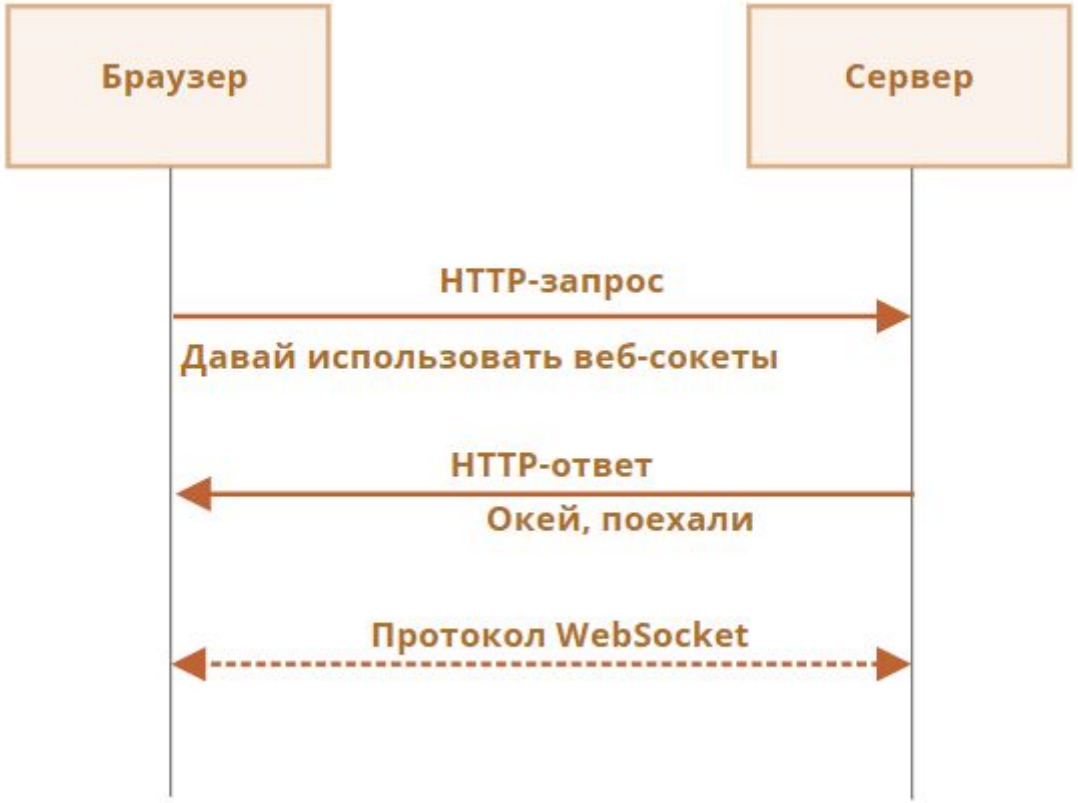
А если мы хотим отправить что-нибудь,
то вызов метода .send(…) сделает это.

```
let socket = new WebSocket("wss://myapplication.herokuapp.com/web");

socket.onopen = function(e) {
  alert("[open] Соединение установлено");
  alert("Отправляем данные на сервер");
  socket.send("Hello server");
};

socket.onmessage = function(event) {
  alert(`[message] Данные получены с сервера: ${event.data}`);
};

socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[close] Соединение закрыто с кодом ${event.code}, ${event.reason}`);
  } else {
    // например, сервер убил процесс или сеть недоступна
    // обычно в этом случае event.code 1006
    alert('[close] Соединение прервано');
  }
};

socket.onerror = function(error) {
  alert(`[error] ${error.message}`);
};
```

Установка WebSocket соединения

Когда `new WebSocket(url)` создан, он тут же сам начинает устанавливать соединение.

Браузер, при помощи специальных заголовков, спрашивает сервер:
«Ты поддерживаешь Websocket?»
и если сервер отвечает «да», они начинают работать по протоколу WebSocket, который уже не является HTTP.



Установка WebSocket соединения

Пример заголовков запроса, который происходит при установке соединения

```
GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

Установка WebSocket соединения

Origin – источник текущей страницы (например https://javascript.info).

Объект WebSocket по своей природе не завязан на текущий источник. Нет никаких специальных заголовков или других ограничений. Старые сервера все равно не могут работать с WebSocket, поэтому проблем с совместимостью нет. Но заголовок Origin важен, так как он позволяет серверу решать, использовать ли WebSocket с этим сайтом.

(Про особенности CORS позже)

Установка WebSocket соединения

Connection: Upgrade
сигнализирует, что клиент хотел бы изменить протокол.

Upgrade: websocket
запрошен протокол «websocket».

Sec-WebSocket-Key: `Iv8io/9s+lYFgZWcXczP8Q==`
случайный ключ, созданный браузером для обеспечения безопасности.

Sec-WebSocket-Version: 13
версия протокола WebSocket, текущая версия 13.

Установка WebSocket соединения

Если сервер согласен переключиться на WebSocket, то он должен отправить в ответ HTTPшный ответ с кодом 101:

```
1  101 Switching Protocols
2  Upgrade: websocket
3  Connection: Upgrade
4  Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

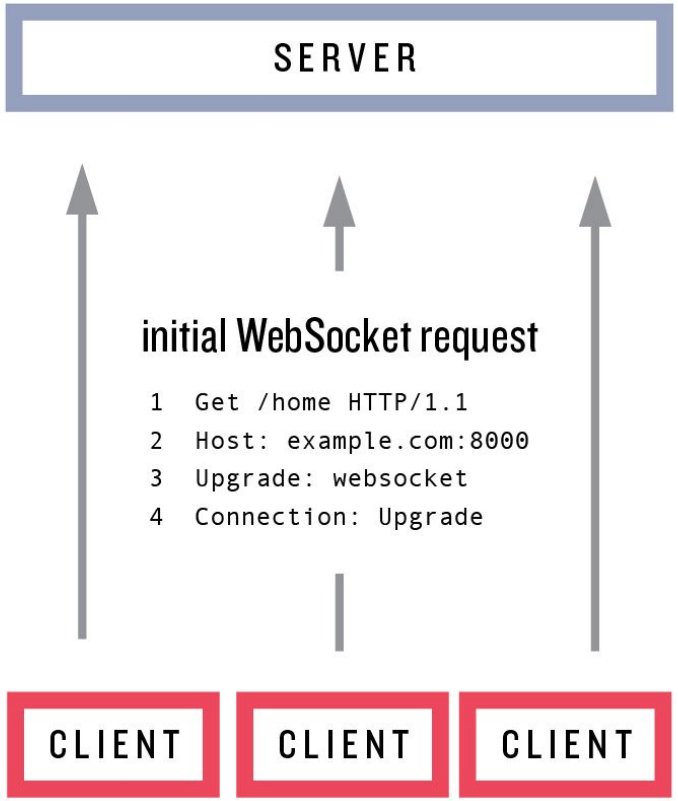
Установка WebSocket соединения

Здесь **Sec-WebSocket-Accept** – это **Sec-WebSocket-Key**, перекодированный с помощью специального алгоритма. Браузер использует его, чтобы убедиться, что ответ соответствует запросу (т.е. ответил тот же сервер, к которому мы обращались).

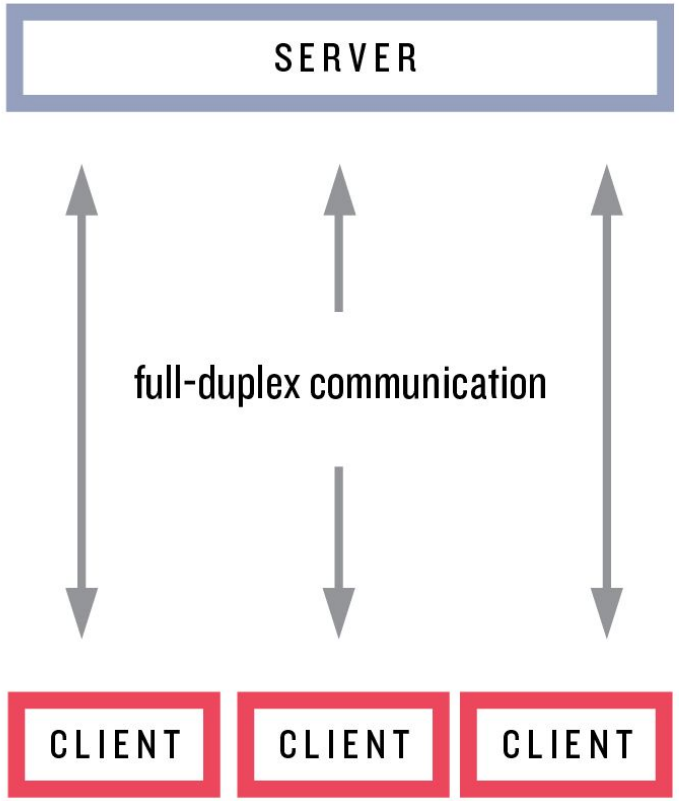
После этого данные передаются по протоколу WebSocket, и вскоре мы увидим его структуру («фреймы»). И это вовсе не HTTP.

WEBSOCKET PROTOCOL

Step 1:
Each client sends a separate request to initiate a WebSocket connection.



Step 2:
The server communicates with each client via a persistent, full-duplex (bi-directional) connection.

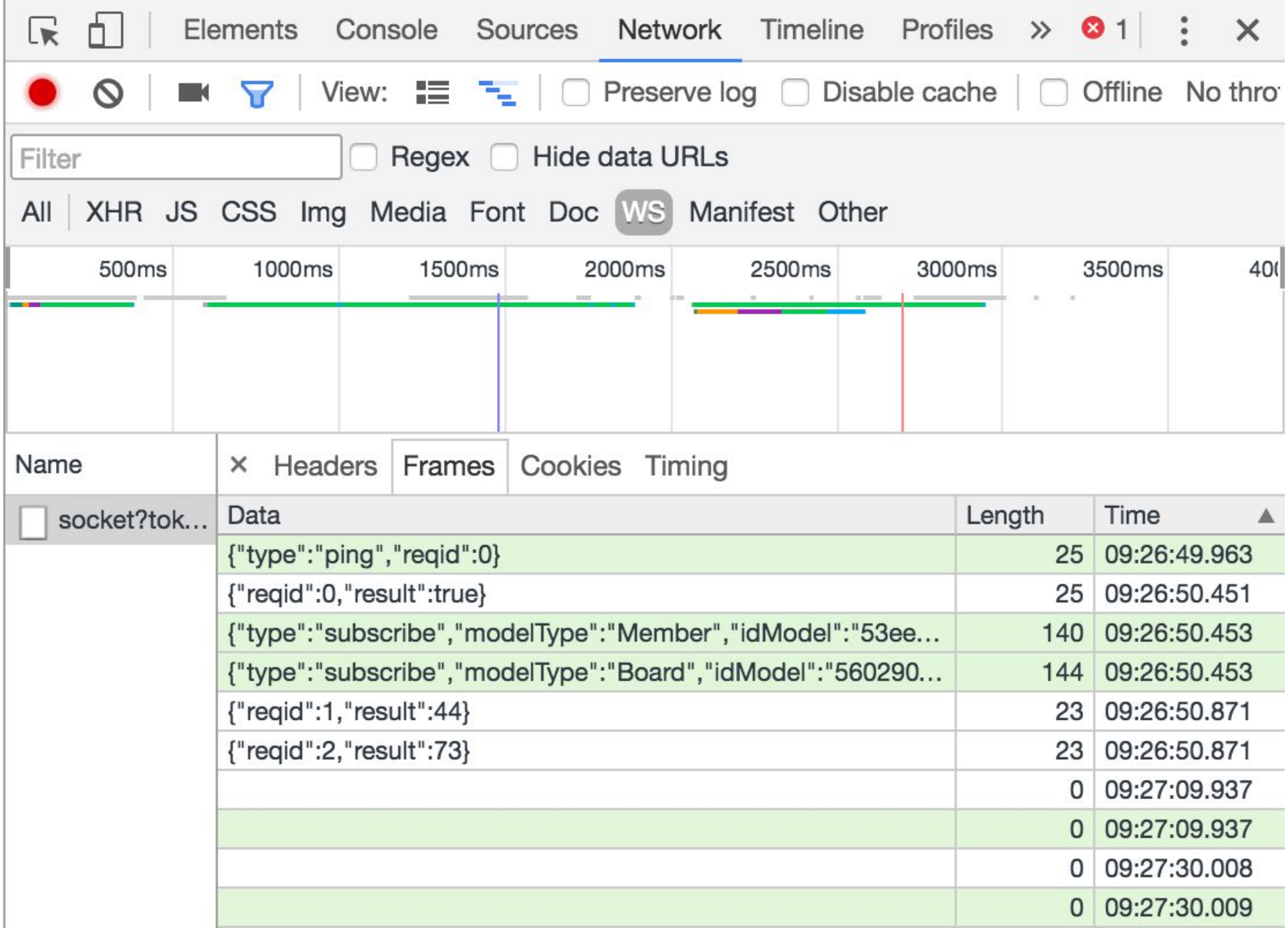


Передача данных в WebSocket

Поток данных в WebSocket состоит из «фреймов», фрагментов данных, которые могут быть отправлены любой стороной, и которые могут быть следующих видов:

- «текстовые фреймы» – содержат текстовые данные, которые стороны отправляют друг другу.
- «бинарные фреймы» – содержат бинарные данные, которые стороны отправляют друг другу.
- «пинг-понг фреймы» используется для проверки соединения; отправляется с сервера, браузер реагирует на них автоматически.
- также есть «фрейм закрытия соединения» и некоторые другие служебные фреймы.

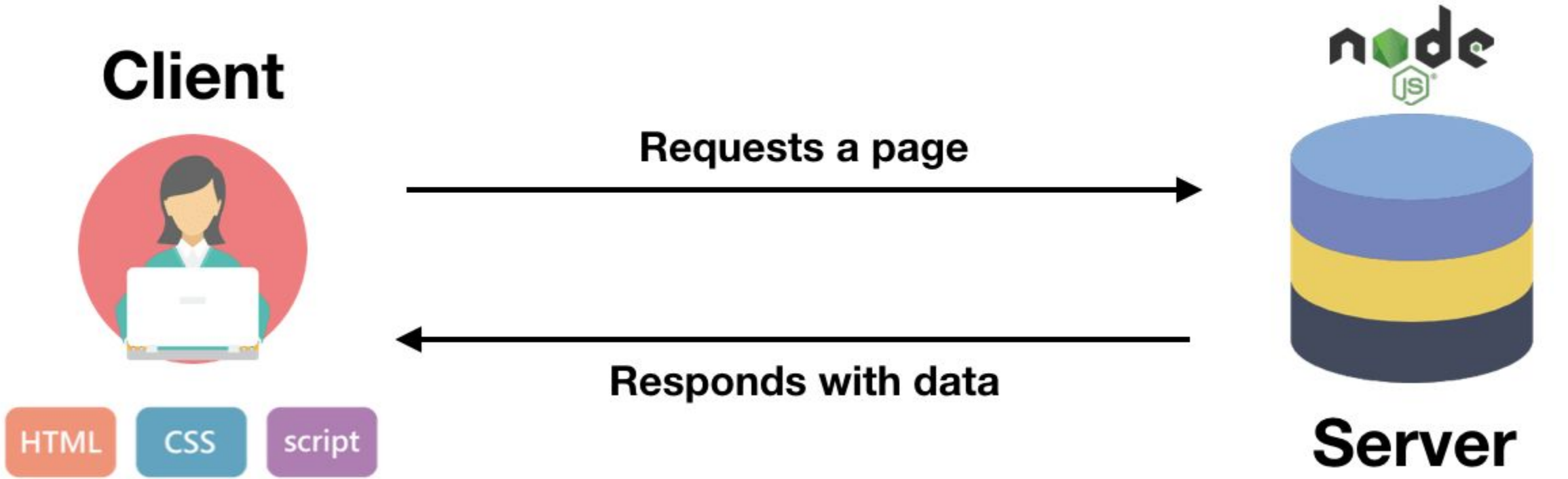
В браузере мы напрямую работаем только с текстовыми и бинарными фреймами.

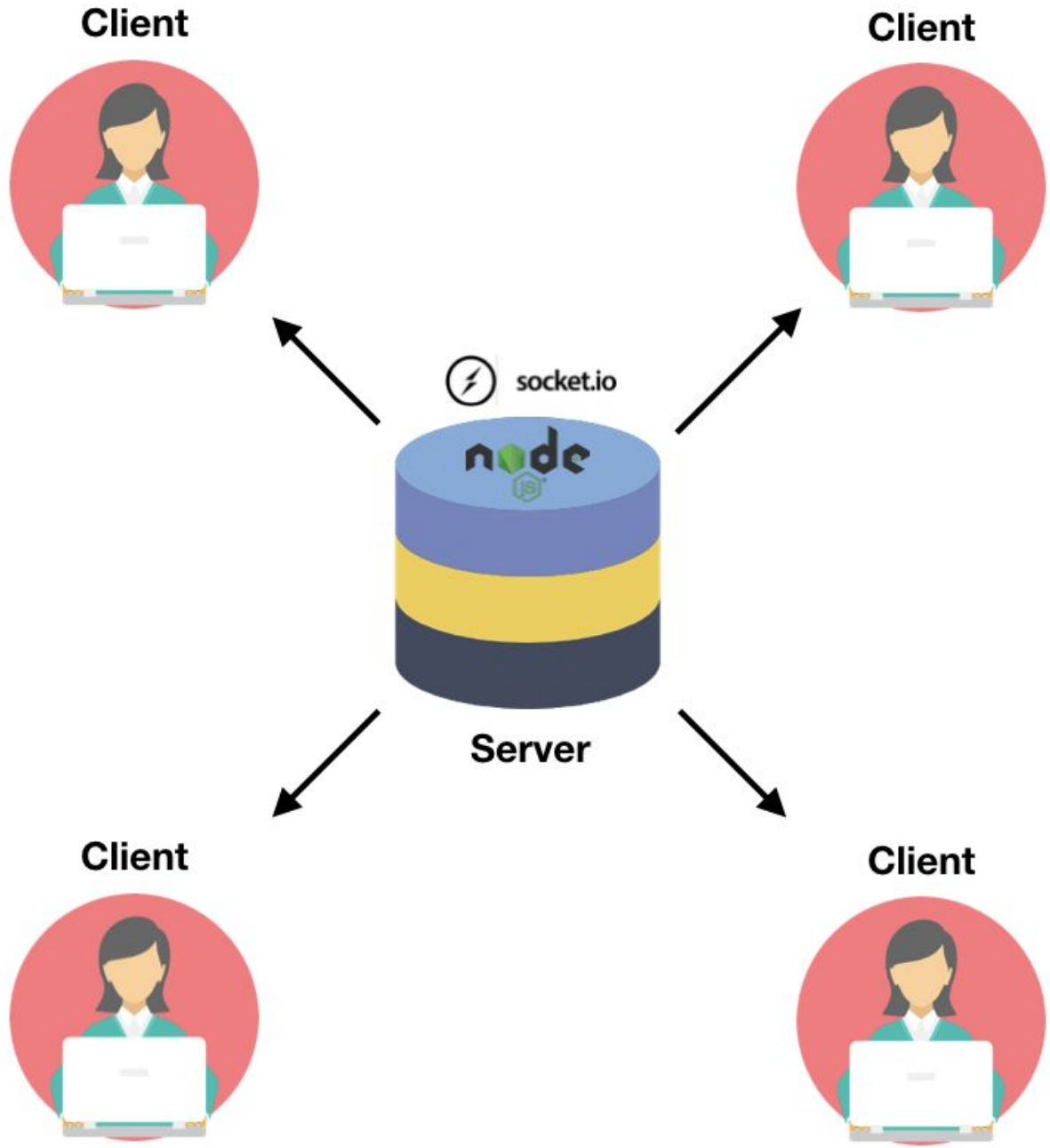

Elements
Console
Sources
Network
Timeline
Profiles
1
View:
Preserve log
Disable cache
Offline
No thro
Filter
Regex
Hide data URLs
All
XHR
JS
CSS
Img
Media
Font
Doc
WS
Manifest
Other
500ms
1000ms
1500ms
2000ms
2500ms
3000ms
3500ms
Name
Headers
Frames
Cookies
Timing
socket?tok...
Data
Length
Time
{"type":"ping","reqid":0}
25
09:26:49.963
{"reqid":0,"result":true}
25
09:26:50.451
{"type":"subscribe","modelType":"Member","idModel":"53ee...
140
09:26:50.453
{"type":"subscribe","modelType":"Board","idModel":"560290...
144
09:26:50.453
{"reqid":1,"result":44}
23
09:26:50.871
{"reqid":2,"result":73}
23
09:26:50.871
0
09:27:09.937
0
09:27:09.937
0
09:27:30.008
0
09:27:30.009

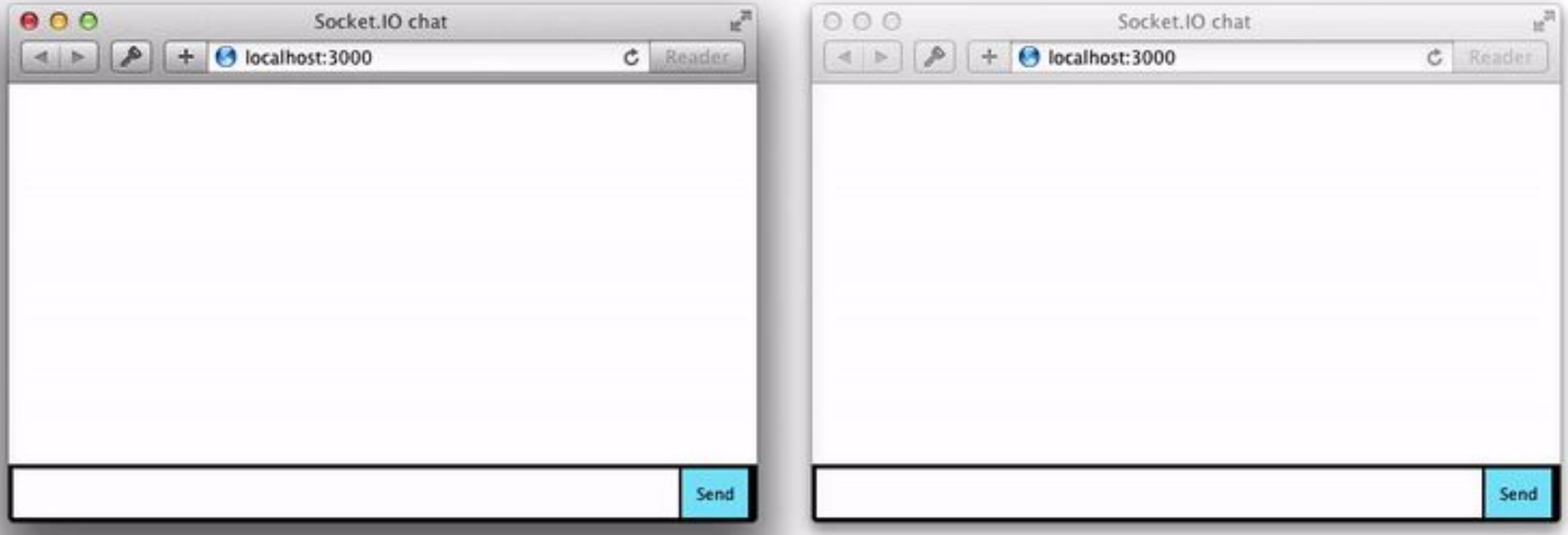
WebSocket Frameworks

WebSocket сам по себе не содержит такие функции, как переподключение при обрыве соединения, аутентификацию пользователей и другие механизмы высокого уровня. Для этого есть клиентские и серверные библиотеки, а также можно реализовать это вручную.

Пакет для нативной работы в сокетами доступен в репозитории
в виде npm пакета “**ws**”


Client
Requests a page
Responds with data
Server
HTML
CSS
script


Client
Client
socket.io
node
Server
Client
Client


Socket.IO chat
localhost:3000
Reader
Send
Socket.IO chat
localhost:3000
Reader
Send

Socket.IO

Socket.IO — библиотека JavaScript, основанная (написанная поверх) на веб-сокетах… и других технологиях. Она использует веб-сокеты, когда они доступны, или такие технологии, как Flash Socket, AJAX Long Polling, AJAX Multipart Stream, когда веб-сокеты недоступны.

Socket.IO

В отличие от веб-сокетов, Socket.IO позволяет отправлять сообщения всем подключенным клиентам. Например, вы пишете чат и хотите уведомлять всех пользователей о подключении нового пользователя. Вы легко можете это реализовать с помощью одной операции. При использовании веб-сокетов, для реализации подобной задачи вам потребуется список подключенных клиентов и отправка сообщений по одному.

Socket.IO

- В веб-сокетах сложно использовать проксирование и балансировщики нагрузки.
  Socket.IO поддерживает эти технологии из коробки.
- Как отмечалось ранее,
  Socket.IO поддерживает постепенную (изящную) деградацию.
- Socket.IO поддерживает автоматическое переподключение при разрыве соединения.
- С Socket.IO легче работать.

Socket.IO

Как настроить socket.io на стороне NodeJS сервера (без фреймворков)

```
const io = require("socket.io")(server);


io.on("connection", socket => {
  console.log("New user connected");
});
```

Socket.IO

Здесь объект io предоставит нам доступ к библиотеке socket.io. Теперь объект io прослушивает каждое соединение с нашим приложением. Каждый раз, когда подключается новый пользователь, он выводит на экран «Новый пользователь подключен».

Если вы попытаетесь перезагрузить наш браузер на localhost, ничего не произойдет … Почему?

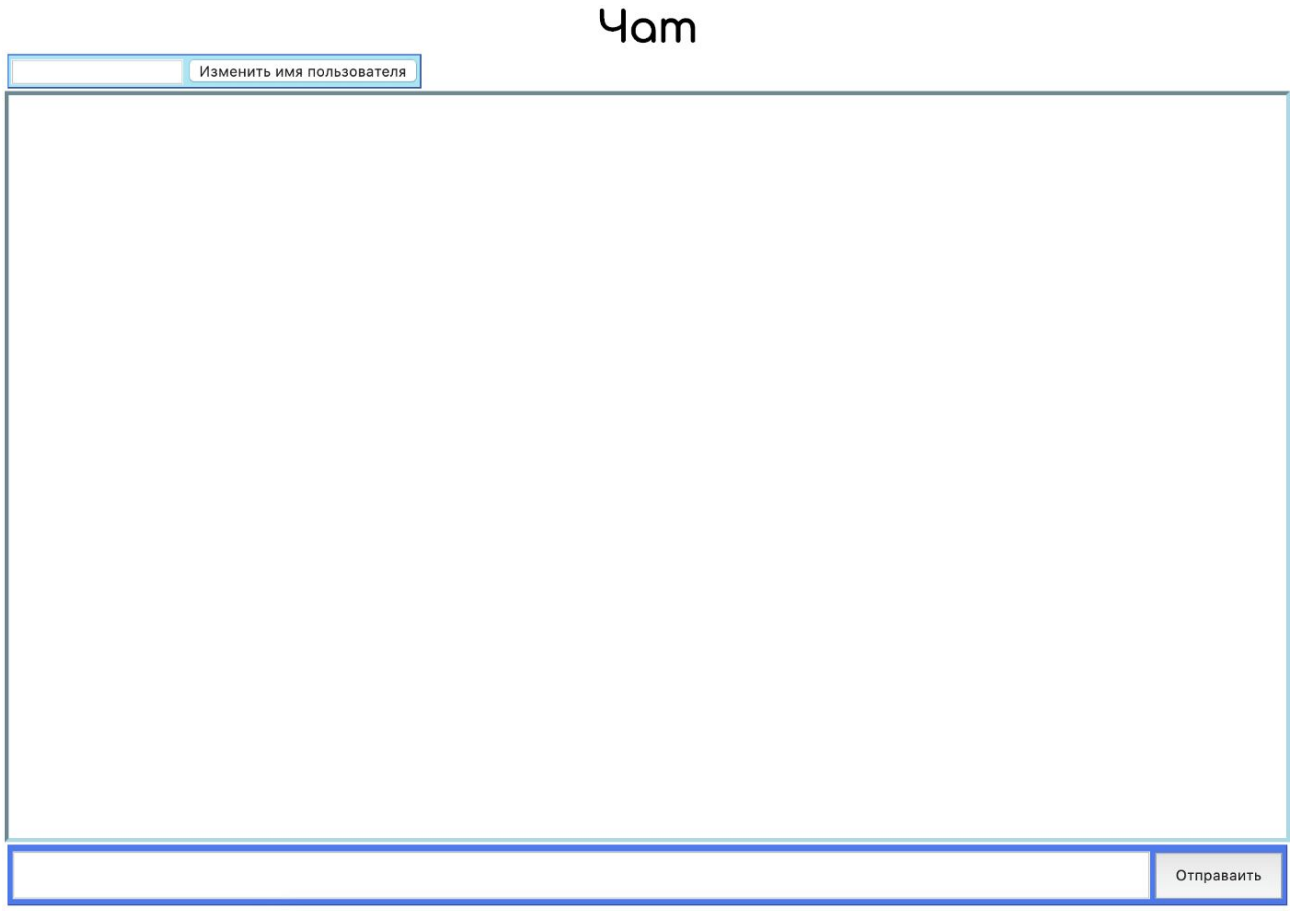
Потому что наша клиентская сторона еще не готова.

Socket.IO

В реальном приложении, для того чтобы всё сработало, нам нужно окно с окном чата, входами для ввода имени пользователя / сообщения и кнопкой отправки. Для этого мы должны отдать html-файл с клиентской частью SocketIO. Давайте сделаем это с помощью Express’a

```
app.get('/', (req, res) => {
    res.render('index')
})
```

Теперь наш localhost:3000 выглядит так:



Socket.IO

Когда у нас есть наш базовый шаблон, мы должны «установить» socket.io на каждом клиенте, который попытается подключиться к нашему серверу. Для этого нам нужно импортировать библиотеку socket.io на стороне клиента:

<script src=”https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.0.4/socket.io.js "></script>

Socket.IO

Далее, необходимо добавить IIFE для инициализации подключения:

let socket = io.connect(“https://localhost:3000”)

Как вы, наверное, догадались, когда клиент загрузит страницу, он автоматически подключится и создаст новый сокет.

Поэтому, когда вы обновите страницу, мы увидим «Новый пользователь подключен» в вашем терминале.

Socket.IO

Когда пользователь подключается к нашему приложению, мы устанавливаем ему / ей имя пользователя по умолчанию, например «анонимный». Для этого нам нужно перейти на серверную часть (app.js) и добавить ключ в сокет. На самом деле, сокет представляет каждого клиента, подключенного к нашему серверу.

```
io.on('connection', (socket) => {
    console.log('New user connected')

    socket.username = "Anonymous"

    socket.on('change_username', (data) => {
        socket.username = data.username
    })
```

Socket.IO

Мы также будем слушать вызовы/события, сделанные в «change_username».
Если на это событие отправлено сообщение, имя пользователя будет изменено.
На стороне клиента цель состоит в том, чтобы сделать наоборот. Каждый раз, когда нажимается кнопка смены имени пользователя, клиент отправляет событие с новым значением.

… для сообщений принцип тот же …

```
$(function(){
  var socket = io.connect('http://localhost:3000')

  var message = $("#message")
  var username = $("#username")
  var send_message = $("#send_message")
  var send_username = $("#send_username")
  var chatroom = $("#chatroom")
  var feedback = $("#feedback")

  send_message.click(function(){
      socket.emit('new_message', {message : message.val()})
  })
```

А что насчёт Socket.IO в Nest’е ?

https://docs.nestjs.com/websockets/gateways

https://github.com/TannerGabriel/Blog/tree/master/NestVueChat

https://is-web-y23-chat.herokuapp.com/