



НЕТОЛОГИЯ

# REST, SERVER SENT EVENTS, WEBSOCKETS



АЛЕКСЕЙ СУДНИЧНИКОВ



# АЛЕКСЕЙ СУДНИЧНИКОВ

Веб-разработчик



[@avsudnichnikov](https://www.telegram.me/avsudnichnikov)



# ПЛАН ЗАНЯТИЯ

1. [fetch](#)
2. [REST](#)
3. [Server Sent Events](#)
4. [WebSockets](#)



# **FETCH & REST API**



## ЗАДАЧА

Создадим небольшое приложение — список клиентов (уникальный идентификатор, имя и телефон). Список должен храниться на сервере и добавление/удаление тоже должно происходить на сервере.

Мы, конечно, можем это организовать с помощью XHR (XMLHttpRequest) и FormData, но давайте посмотрим на современные альтернативы.



# FETCH

# FETCH

Fetch API – новое API, построенное на `Promise`, для упрощения взаимодействия по HTTP:

```
const response = await fetch(<url>);
```

Общий формат:

```
1  fetch(<url>, { <- RequestInit
2    body: <body>,
3    method: <method>,
4    headers: {
5      <key>: <value>
6    }
7  })
```

# REQUESTINIT

```
1 dictionary RequestInit {  
2     ByteString method;  
3     HeadersInit headers;  
4     BodyInit? body;  
5     USVString referrer;  
6     ReferrerPolicy referrerPolicy;  
7     RequestMode mode;  
8     RequestCredentials credentials;  
9     RequestCache cache;  
10    RequestRedirect redirect;  
11    DOMString integrity;  
12    boolean keepalive;  
13    AbortSignal? signal;  
14    any window; // can only be set to null  
15 };
```



# BODYINIT

```
1  typedef (Blob or
2      BufferSource or FormData or
3      URLSearchParams or ReadableStream or USVString
4  ) BodyInit;
```

Т.е. в теле запроса могут отправляться `ArrayBuffer`, `Blob`, строковые данные, `URLSearchParams`, `FormData`.

# FETCH

Ответ возвращает объект типа `Response` :

```
1 interface Response {
2   [NewObject] static Response error();
3   [NewObject] static Response redirect(USVString url, optional unsigned short status = 302);
4
5   readonly attribute ResponseType type;
6
7   readonly attribute USVString url;
8   readonly attribute boolean redirected;
9   readonly attribute unsigned short status;
10  readonly attribute boolean ok;
11  readonly attribute ByteString statusText;
12  [SameObject] readonly attribute Headers headers;
13  readonly attribute Promise<Headers> trailer;
14
15  [NewObject] Response clone();
16 };
```

# FETCH

`Response` содержит mixin `Body`:

```
1 interface mixin Body {  
2     readonly attribute ReadableStream? body;  
3     readonly attribute boolean bodyUsed;  
4     [NewObject] Promise<ArrayBuffer> arrayBuffer();  
5     [NewObject] Promise<Blob> blob();  
6     [NewObject] Promise<FormData> formData();  
7     [NewObject] Promise<any> json();  
8     [NewObject] Promise<USVString> text();  
9 };
```

# FETCH

Типичный сценарий использования:

```
1  const response = await fetch('<url>');  
2  if (response.ok) {  
3      const data = await response.json();  
4  }
```



# JSON

Q: Почему JSON?

A: Потому что легковесен, поддерживается почти везде (mobile/web/etc) и прост в отладке (т.к. текстовый).



# REST



# REST

Принцип построения API (где-то можно встретить слова про архитектуру, распределённые системы и т.д.).



# REST

Несмотря на то, что определение было дано товарищем Roy Fielding, сейчас оно настолько «расплылось», что мы выделим для себя лишь основные критерии:

1. Клиент-серверное взаимодействие
2. URL'ы — как иерархические наборы ресурсов (получается не всегда)
3. Методы HTTP — как определяющие действия:
  - GET — получить
  - POST/PUT — создать/обновить
  - DELETE — удалить
4. В качестве контейнера для данных используется чаще всего JSON



# GITHUB API

В качестве примера рассмотрим GitHub API:

<https://developer.github.com/v3/repos>

В частности, получение информации о репозиториях:

```
1 GET /user/repos - своих
2 GET /users/:owner/repos - конкретного пользователя
3 GET /repos/:owner/:repo - информации о конкретном
```

Создание репозитория:

```
1 POST /user/repos
2 {
3   "name": "Hello-World",
4   "description": "This is your first repository",
5   "homepage": "https://github.com",
6   "private": false,
7   "has_issues": true,
8   "has_projects": true,
9   "has_wiki": true
10 }
```

# GITHUB API

Изменение репозитория:

```
1 PATCH /repos/:owner/:repo
2 {
3   "name": "Hello-World",
4   "description": "This is your first repository",
5   "homepage": "https://github.com",
6   "private": true,
7   "has_issues": true,
8   "has_projects": true,
9   "has_wiki": true
10 }
```

Отображение тегов (дочерний ресурс):

```
GET /repos/:owner/:repo/tags
```

Удаление репозитория:

```
DELETE /repos/:owner/:repo
```



# REST

Таким образом, выстраивается некая достаточно простая система взаимодействия, когда frontend и backend обмениваются данными в формате JSON, а URL'ы определяют, с какими ресурсами мы работаем.



## ВСПОМИНАЕМ ЗАДАЧУ

Небольшое приложение — список клиентов (уникальный идентификатор, имя и телефон). Список должен храниться на сервере и добавление/удаление тоже должно происходить на сервере.

# ЗАДАЧА

Создадим небольшую обёртку над `fetch`:

```
1 export default class API {
2   constructor(url) {
3     this.url = url;
4     this.contentTypeHeader = {'Content-Type': 'application/json'};
5   }
6
7   load() {
8     return fetch(this.url);
9   }
10
11  add(contact) {
12    return fetch(this.url, {
13      body: JSON.stringify(contact),
14      method: 'POST',
15      headers: this.contentTypeHeader,
16    });
17  }
18
19  remove(id) {
20    return fetch(`${this.url}/${id}`, {
21      method: 'DELETE'
22    });
23  }
24 }
```



# REST API

В качестве основы возьмём проект с предыдущей лекции со следующими зависимостями:

```
npm install koa koa-body
```

# REST API

Мы, конечно, можем на сервере вручную разбирать метод и путь (да ещё оттуда вытаскивать параметры, вроде `id` для `DELETE`).

Но лучше воспользуемся готовым решением `koa-router`:

```
npm install koa-router
```

```
1  const Router = require('koa-router');
2  const router = new Router();
3  const contacts = [];
4
5  app.use(koaBody({
6    urlencoded: true,
7    multipart: true,
8    json: true,
9  }));
10
11 router.get('/contacts', async (ctx, next) => {
12   // return list of contacts
13   ctx.response.body = contacts;
14 });
15 router.post('/contacts', async (ctx, next) => {
16   // create new contact
17   contacts.push({...ctx.request.body, id: uuid.v4()});
18   ctx.response.status = 204;
19 });
20 router.delete('/contacts/:id', async (ctx, next) => {
21   // remove contact by id (ctx.params.id)
22   const index = contacts.findIndex(({ id }) => id === ctx.params.id);
23   if (index !== -1) {
24     contacts.splice(index, 1);
25   };
26   ctx.response.status = 204;
27 });
28
29 app.use(router.routes());
30 app.use(router.allowedMethods());
```



# REST CLIENT

```
1  const api = new API('http://localhost:7070/contacts');
2  {
3    const response = await api.load();
4    const data = await response.json();
5    console.log(data);
6  }
7  {
8    const response = await api.add({name: 'Ivan', phone: '+79.....'});
9  }
10 {
11   const response = await api.load();
12   const data = await response.json();
13   console.log(data);
14 }
```

Примечание\*: отрисовку в DOM и взаимодействие с формой для краткости мы не рассматриваем, поскольку это вы уже знаете.



# SERVER SENT EVENTS



# ЗАДАЧА

Перед нами поставили задачу: реализовать интерфейс для сервиса, умеющего динамически подгружать обновления: «горячие новости», новые фото и видео в ленте новостей.

Вопрос к аудитории: какие есть предложения по решению данной задачи?



# AJAX

Первой мыслью чаще всего будет использование AJAX (XHR или fetch) вкупе с `setTimeout/setInterval` (надеюсь вы не предложили идею с кнопкой «Обновить»).

В большинстве случаев именно она будет выбрана с точки зрения простоты реализации, поддерживаемости и других параметров.

# ЧАСТОТА ОБНОВЛЕНИЯ

Но есть ключевой вопрос: с какой частотой посылать запросы?

Это ведь зависит достаточно сильно от интенсивности появления событий, причём если контент генерируется самими пользователями, то предсказать частоту появления можно будет только с помощью догадок, статистики или инструментов Machine Learning.

Если это начать учитывать в логике, то, получается, мы должны от сервера получать информацию о том, через сколько слать следующий запрос (а что если в это время что-то произойдёт?).

# SSE

Давайте посмотрим на альтернативы. Стандарт HTML предлагает нам технологию [Server Sent Events](#).

Поддержка не полная, но использовать можно:

Current aligned	Usage relative	Date relative	Apply filters	Show all	?				
IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
		64	71						
	17	65	72		11.4				4
11	18	66	73	12	12.1	all	71	11.8	8.2
		67	74	12.1	12.2				
		68	75	TP					
			76						

```

1  if (!window.EventSource) {
2    // fallback to xhr
3    return;
4  }

```



# SSE

Server Sent Events — это технология, которая позволяет от сервера клиенту отправлять уведомления через открытое соединение.

Обратите внимание: уведомление может отправлять только сервер. При этом сам сервер решает, с какой частотой их слать.

# EVENTSOURCE

Стандарт HTML Предлагает нам объект `EventSource`, который и позволяет организовывать соединение с сервером

```
1  [Constructor(USVString url, optional EventSourceInit eventSourceInitDict)]
2  interface EventSource : EventTarget {
3      readonly attribute USVString url;
4      readonly attribute boolean withCredentials;
5
6      // ready state
7      const unsigned short CONNECTING = 0;
8      const unsigned short OPEN = 1;
9      const unsigned short CLOSED = 2;
10     readonly attribute unsigned short readyState;
11
12     // networking
13     attribute EventHandler onopen;
14     attribute EventHandler onmessage;
15     attribute EventHandler onerror;
16     void close();
17 };
```

Т.е. всё, что нужно — это указать URL для подключения при создании объекта. Кроме того, поскольку объект является `EventTarget` 'ом, мы можем подписываться на события, либо использовать `EventHandler` 'ы.

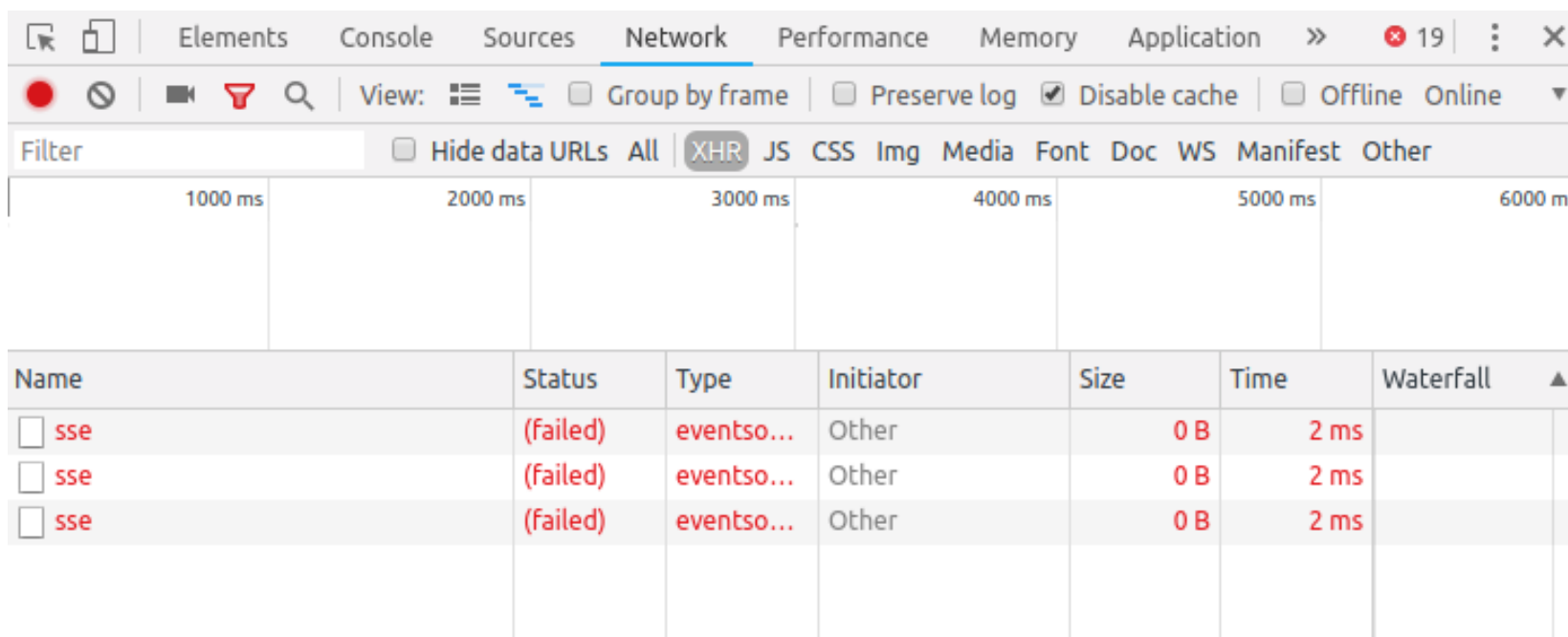


# EVENTSOURCE

```
1  const eventSource = new EventSource("http://localhost:7070/sse");
2  eventSource.addEventListener('message', (evt) => {
3    console.log(evt);
4  });
5  eventSource.addEventListener('open', (evt) => {
6    console.log('connected');
7  });
8  eventSource.addEventListener('error', (evt) => {
9    console.log('error');
10 });
```

# EVENTSOURCE

При отсутствующем сервере запросы будут завершаться с ошибкой:



The screenshot shows the Chrome DevTools Network tab with the 'XHR' filter selected. The timeline at the top shows a 6000 ms duration. Below the timeline, a table lists three failed SSE requests. Each request is marked as '(failed)' and has a size of 0 B and a time of 2 ms. The initiator for all three is 'Other'.

Name	Status	Type	Initiator	Size	Time	Waterfall
<input type="checkbox"/> sse	(failed)	eventso...	Other	0 B	2 ms	
<input type="checkbox"/> sse	(failed)	eventso...	Other	0 B	2 ms	
<input type="checkbox"/> sse	(failed)	eventso...	Other	0 B	2 ms	

Но если мы посмотрим, окажется, что браузер автоматически пытается переподключиться.

# PROCESS MODEL

Процесс описан в разделе `Process model` [спецификации](#):

1. При создании объекта `EventSource` запрашивается URL
2. Ожидается ответ `200 OK` и `Content-Type: text/event-stream`
3. После чего осуществляется приём данных

Если произошла сетевая ошибка (как в нашем случае), то браузер будет пытаться переподключиться самостоятельно.

Если в ответ на запрос при подключении приходит не `200 OK` `Content-Type: text/event-stream`, то браузер не пытается переподключиться.

# СЕРВЕР

Поскольку нам нужно куда-то подключаться, нужен сервер. Создадим его на базе Koa:

```
npm init
npm install koa koa-router http-event-stream uuid forever
```

Файл `.foreverignore`:

```
node_modules
public
```

Скрипты:

```
1  "scripts": {
2    "watch": "forever -w server.js",
3    "prestart": "npm install",
4    "start": "forever server.js"
5  },
```

# ЗАВИСИМОСТИ

```
1  const http = require('http');  
2  const Koa = require('koa');  
3  const { streamEvents } = require('http-event-stream');  
4  const uuid = require('uuid');  
5  const app = new Koa();
```

# CORS

Несмотря на то, что про SOP и CORS ничего не было сказано, без CORS мы не получим 200 OK:

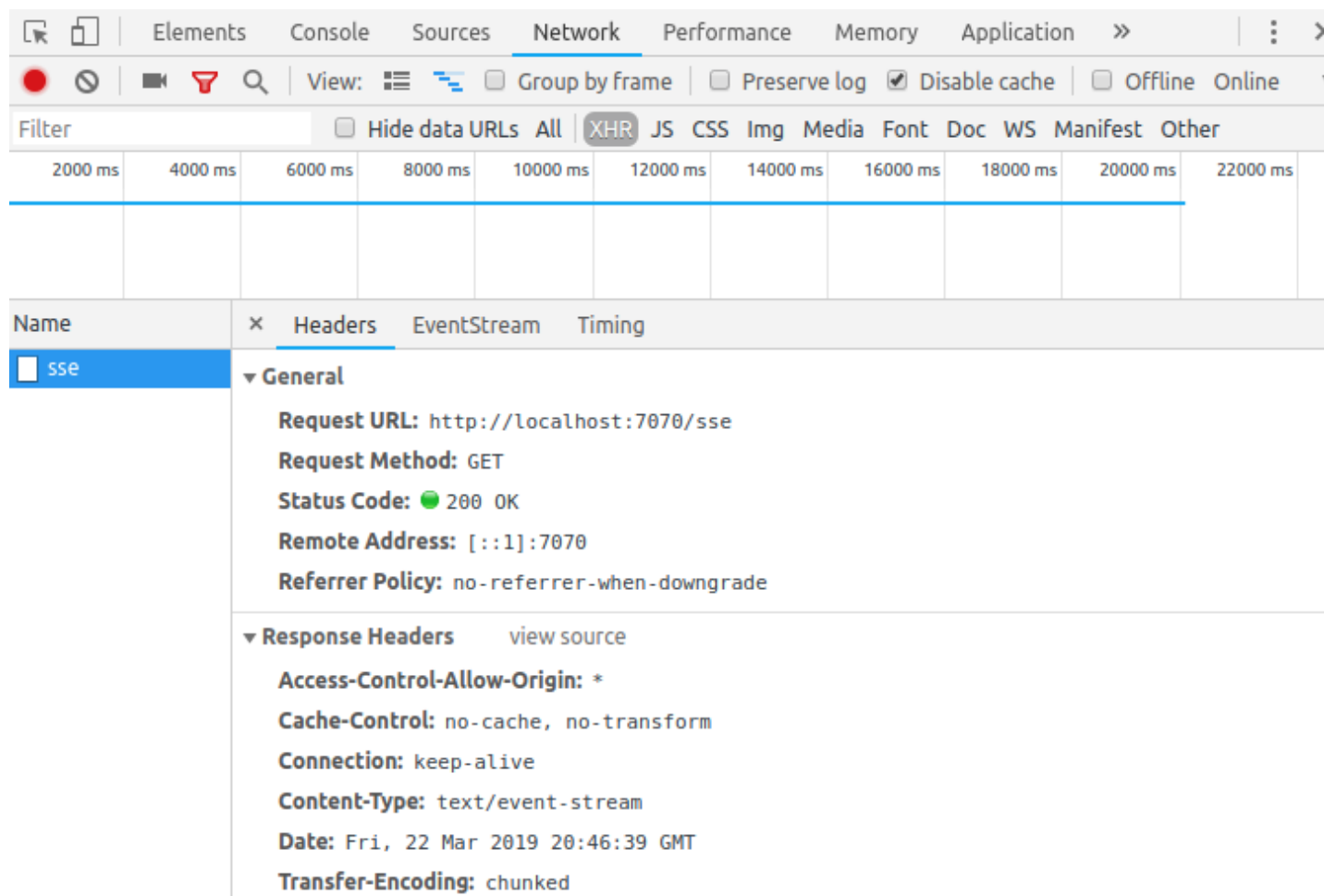
```
1 app.use(async (ctx, next) => {
2   const origin = ctx.request.get('Origin');
3   if (!origin) {
4     return await next();
5   }
6
7   const headers = { 'Access-Control-Allow-Origin': '*', };
8
9   if (ctx.request.method !== 'OPTIONS') {
10    ctx.response.set({ ...headers });
11    try {
12      return await next();
13    } catch (e) {
14      e.headers = { ...e.headers, ...headers };
15      throw e;
16    }
17  }
18
19  if (ctx.request.get('Access-Control-Request-Method')) {
20    ctx.response.set({
21      ...headers,
22      'Access-Control-Allow-Methods': 'GET, POST, PUD, DELETE, PATCH',
23    });
24
25    if (ctx.request.get('Access-Control-Request-Headers')) {
26      ctx.response.set('Access-Control-Allow-Headers', ctx.request.get('Access-Control-Request-Headers'));
27    }
28
29    ctx.response.status = 204;
30  }
31 });
```

# ROUTER

Поскольку мы сразу хотим, чтобы обрабатывался только определённый URL, а не все, подключим SSE с помощью `koa-router`:

```
1  const Router = require('koa-router');
2  const router = new Router();
3
4  router.get('/sse', async (ctx) => {
5    streamEvents(ctx.req, ctx.res, {
6      async fetch(lastEventId) {
7        console.log(lastEventId);
8        return [];
9      },
10     stream(sse) {
11       sse.sendEvent({data: 'hello world'});
12
13       return () => {};
14     }
15   });
16
17   ctx.respond = false; // коа не будет обрабатывать ответ
18 });
19
20 app.use(router.routes()).use(router.allowedMethods());
21
22 const port = process.env.PORT || 7070;
23 const server = http.createServer(app.callback()).listen(port)
```

# SSE

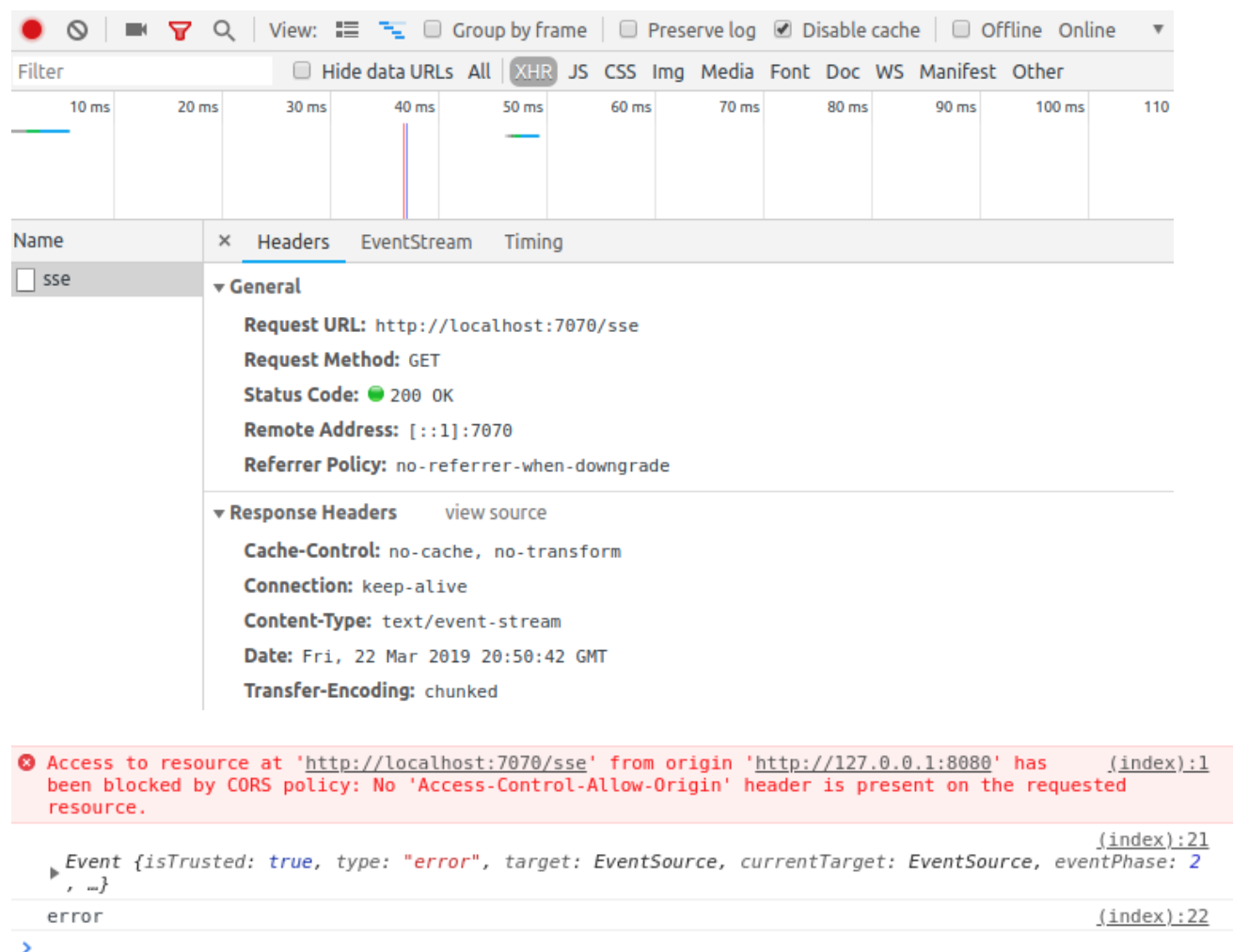


Видно, что при работающем сервере и включенном CORS подключение устанавливается один раз и не закрывается (см. Timeline).



# CORS

При выключенном CORS:



The screenshot displays the Chrome DevTools interface. At the top, the 'Network' tab is active, showing a list of requests. The 'XHR' filter is selected, and a single request to 'http://localhost:7070/sse' is visible. The 'Headers' sub-tab is open, showing the following details:

- General:**
  - Request URL: `http://localhost:7070/sse`
  - Request Method: `GET`
  - Status Code: `200 OK`
  - Remote Address: `::1:7070`
  - Referrer Policy: `no-referrer-when-downgrade`
- Response Headers:**
  - Cache-Control: `no-cache, no-transform`
  - Connection: `keep-alive`
  - Content-Type: `text/event-stream`
  - Date: `Fri, 22 Mar 2019 20:50:42 GMT`
  - Transfer-Encoding: `chunked`

Below the headers, the console shows an error event:

```
Event {isTrusted: true, type: "error", target: EventSource, currentTarget: EventSource, eventPhase: 2, ...}
```

The console also displays the following error message:

```
Access to resource at 'http://localhost:7070/sse' from origin 'http://127.0.0.1:8080' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

# HTTP-EVENT-STREAM

Давайте разбираться с пакетом `http-event-stream`:

```
1 streamEvents(ctx.req, ctx.res, {  
2   ...  
3 });  
4  
5 ctx.respond = false; // коа не будет обрабатывать ответ
```

- `ctx.req`, `ctx.res` – нативные запрос/ответ Node.js (не обёртки от Koa)
- `ctx.respond = false` – указание коа на то, что обработку ответа мы берём на себя

# STREAMEVENTS

Функция, которая и осуществляет всю работу — содержит объект `EventStreamOptions`:

```
1 export interface EventStreamOptions {  
2     /** How often to send a keep-alive comment. In milliseconds. */  
3     keepAliveInterval?: number;  
4     fetch(lastEventId: string): Promise<ServerSentEvent[]>;  
5     stream(context: StreamContext): UnsubscribeFn;  
6     onError?(error: Error): void;  
7 }
```

По факту:

- `fetch` — удобная функция для отправки клиенту пула «непрочитанных сообщений»
- `stream` — основная функция, в которой происходит отправка сообщений, должна возвращать функцию для `unsubscribe`

## НА ПРИМЕРЕ

```
1  streamEvents(ctx.req, ctx.res, {
2    async fetch(lastEventId) {
3      console.log(lastEventId);
4      return [];
5    },
6    stream(sse) {
7      const interval = setInterval(() => {
8        sse.sendEvent({ data: 'hello world' });
9      }, 5000);
10
11      return () => clearInterval(interval);
12    }
13  });
```

Всё просто. `http-event-stream` самостоятельно обрабатывает отключения клиентов и ошибки.

# СТРУКТУРА СООБЩЕНИЙ

На самом деле отправляемые сообщения могут содержать следующие поля:

```
1  export interface ServerSentEvent {  
2      data: string | string[];  
3      event?: string;  
4      id?: string;  
5      retry?: number;  
6  }
```

# RAW

В «чистом» виде выглядит это примерно так:

```
1 data: Some text
2
3 data: Some text // <- next message
4
5 data: First line // <- multiline message
6 data: Second line
7
8 id: uuid
9 event: custom
10 data: some data
```

# СТРУКТУРА СООБЩЕНИЙ

- **data** – строка или массив строк (на самом деле это http-event-stream разбивает массив на строки)
- **event** – тип события
- **id** – id события

```
1 stream(sse) {  
2   const interval = setInterval(() => {  
3     sse.sendEvent({  
4       id: uuid.v4(),  
5       data: JSON.stringify({field: 'value'}),  
6       event: 'comment'  
7     });  
8   }, 5000);  
9 }
```

Name	×	Headers	EventStream	Timing
<input checked="" type="checkbox"/> sse		Id	Type	Data
		76dbffc9-05...	comment	{"field": "value"}
		f1ef31bd-d9...	comment	{"field": "value"}
				Time
				00:09:29.977
				00:09:34.980

Но при этом мы перестанем получать события в обработчике **message** на клиенте.

# EVENT

Поле `event` позволяет гибко настраивать тип события:

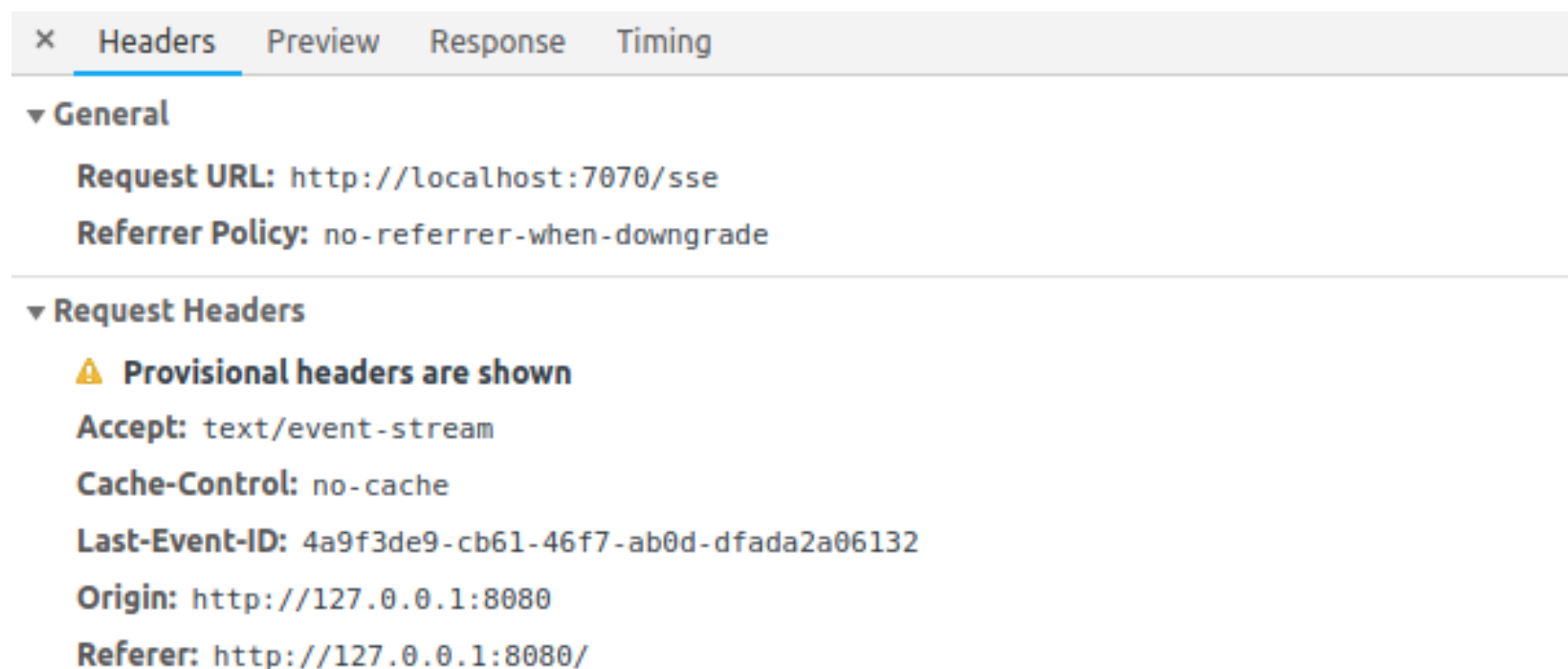
```
1 | eventSource.addEventListener('comment', (evt) => {  
2 |     console.log(evt);  
3 | });
```

Таким образом, сервер может нам сообщать о разных событиях.



# LAST EVENT ID

Если мы выставляли `id` у сообщений, то браузер при обрыве связи и последующем переподключении будет отсылать заголовок `Last-Event-ID`:



× Headers Preview Response Timing

▼ General

**Request URL:** http://localhost:7070/sse

**Referrer Policy:** no-referrer-when-downgrade

---

▼ Request Headers

⚠ Provisional headers are shown

**Accept:** text/event-stream

**Cache-Control:** no-cache

**Last-Event-ID:** 4a9f3de9-cb61-46f7-ab0d-dfada2a06132

**Origin:** http://127.0.0.1:8080

**Referer:** http://127.0.0.1:8080/

# LAST EVENT ID

Именно он в качестве значения придёт в `fetch`:

```
1  async fetch(lastEventId) {  
2    console.log(lastEventId);  
3    return [];  
4  },
```

Примечание\*: для тестирования обрыва связи не обязательно что-то делать с сервером, достаточно выставить флажок `Offline` в Developer Tools.



## ИТОГИ

SSE — достаточно удобный механизм, позволяющий при малом количестве затрат организовать получение обновлений данных с сервера.

Дело за малым — организовать взаимодействие с DOM, но это вы уже умеете.



# WEBSOCKETS



# WEBSOCKETS

Другая задача — организация онлайн-чата.

Опять-таки здесь возможна куча вариантов, начиная от XHR (fetch), заканчивая комбинацией XHR (fetch) для отправки сообщений, а SSE для получения обновлений.

Но эту же задачу можно решить с помощью инструмента, который позволяет организовать двустороннюю связь (в отличие от SSE), когда при открытом соединении и сервер, и клиент могут отсылать данные.

# WEBSOCKETS

Сам протокол описан в стандарте [IETF](#), а предоставляемое API в спецификации [WHATWG](#)

Поддержка:

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
		64	71						
	17	65	72		11.4				4
11	18	66	73	12	12.1	all	71	11.8	8.2
		67	74	12.1	12.2				
		68	75	TP					
			76						

# API

```
1 [Constructor(USVString url, optional (DOMString or sequence<DOMString>) protocols = [])]
2 interface WebSocket : EventTarget {
3     readonly attribute USVString url;
4
5     // ready state
6     const unsigned short CONNECTING = 0;
7     const unsigned short OPEN = 1;
8     const unsigned short CLOSING = 2;
9     const unsigned short CLOSED = 3;
10    readonly attribute unsigned short readyState;
11    readonly attribute unsigned long long bufferedAmount;
12
13    // networking
14    attribute EventHandler onopen;
15    attribute EventHandler onerror;
16    attribute EventHandler onclose;
17    readonly attribute DOMString extensions;
18    readonly attribute DOMString protocol;
19    void close(optional [Clamp] unsigned short code, optional USVString reason);
20
21    // messaging
22    attribute EventHandler onmessage;
23    attribute BinaryType binaryType;
24    void send(USVString data);
25    void send(Blob data);
26    void send(ArrayBuffer data);
27    void send(ArrayBufferView data);
28 };
```

# API

Интерфейс «похож» на `EventSource` и позволяет работать следующим образом:

```
1  const ws = new WebSocket('ws://localhost:7070/ws');
2  ws.binaryType = 'blob'; // arraybuffer
3
4  ws.addEventListener('open', () => {
5    console.log('connected');
6    // After this we can send messages
7    ws.send('hello!');
8  });
9  ws.addEventListener('message', (evt) => {
10    // handle evt.data
11    console.log(evt);
12  });
13 ws.addEventListener('close', (evt) => {
14   console.log('connection closed', evt);
15   // After this we can't send messages
16 });
17 ws.addEventListener('error', () => {
18   console.log('error');
19 });
```



# ОТПРАВКА СООБЩЕНИЙ

При отправке сообщений, если мы находимся в состоянии `CONNECTING`, по спецификации будет выброшен `Exception`. А вот в других состояниях — нет, поэтому при отправке стоит проверять состояние:

```
1  if (ws.readyState === WebSocket.OPEN) {  
2      ws.send(...);  
3  } else {  
4      // Reconnect  
5  }
```

# CLOSE CODE

В событии close содержится поле code, которое позволяет строить дальнейшую логику.

Status Code	Meaning	Contact	Reference
1000	Normal Closure	hybi@ietf.org	<a href="#">RFC 6455</a>
1001	Going Away	hybi@ietf.org	<a href="#">RFC 6455</a>
1002	Protocol error	hybi@ietf.org	<a href="#">RFC 6455</a>
1003	Unsupported Data	hybi@ietf.org	<a href="#">RFC 6455</a>
1004	---Reserved----	hybi@ietf.org	<a href="#">RFC 6455</a>
1005	No Status Rcvd	hybi@ietf.org	<a href="#">RFC 6455</a>
1006	Abnormal Closure	hybi@ietf.org	<a href="#">RFC 6455</a>
1007	Invalid frame payload data	hybi@ietf.org	<a href="#">RFC 6455</a>
1008	Policy Violation	hybi@ietf.org	<a href="#">RFC 6455</a>
1009	Message Too Big	hybi@ietf.org	<a href="#">RFC 6455</a>
1010	Mandatory Ext.	hybi@ietf.org	<a href="#">RFC 6455</a>
1011	Internal Server Error	hybi@ietf.org	<a href="#">RFC 6455</a>
1015	TLS handshake	hybi@ietf.org	<a href="#">RFC 6455</a>

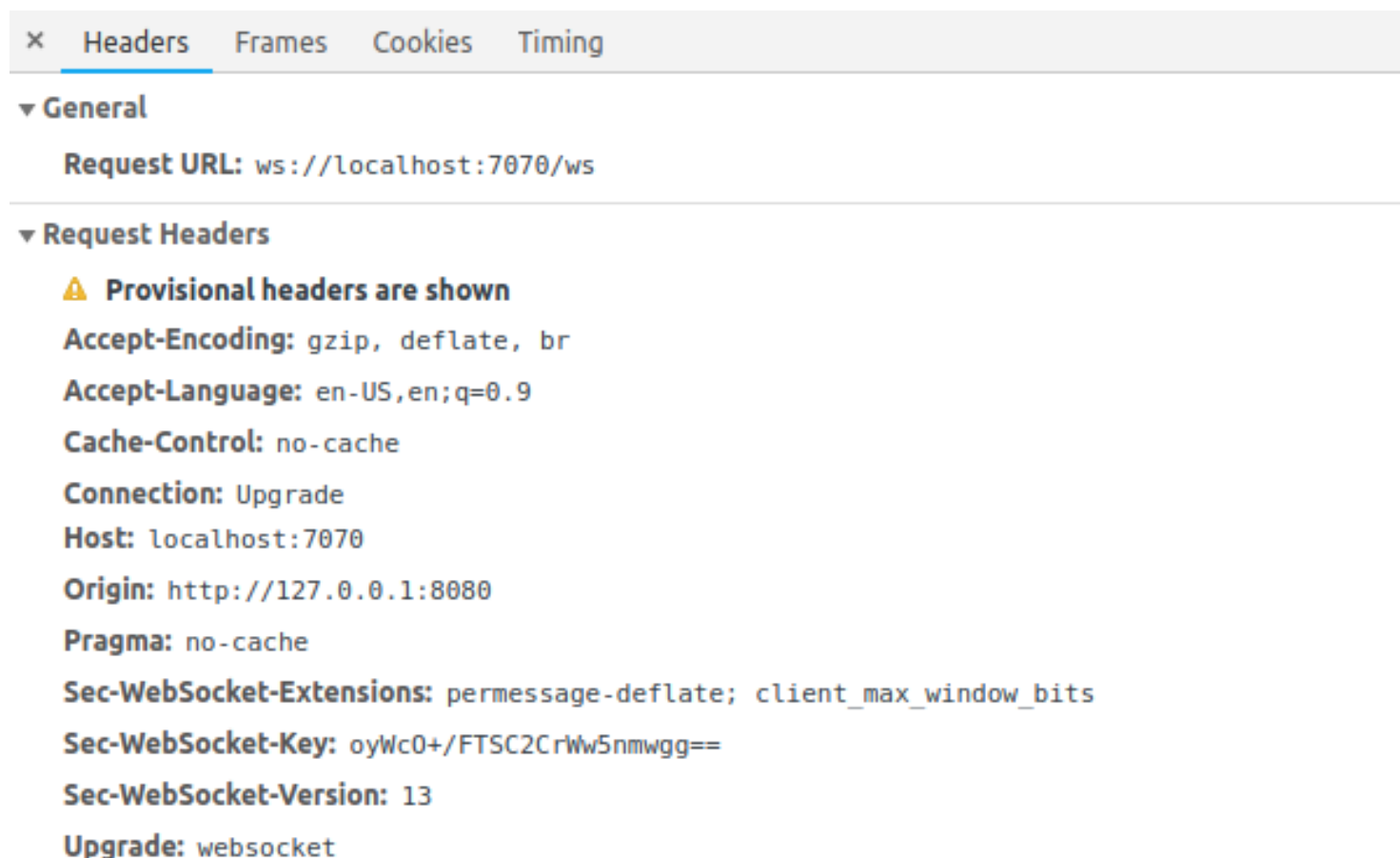
# CLOSE CODE

При подключении к нашему серверу мы получим код 1006:

```
✖ ▶WebSocket connection to 'ws://localhost:7070/ws' failed: Connection closed before receiving a handshake response (index):26
error (index):37
connection closed (index):34
▶CloseEvent {isTrusted: true, wasClean: false, code: 1006, reason: "", type: "close", ...}
> |
```

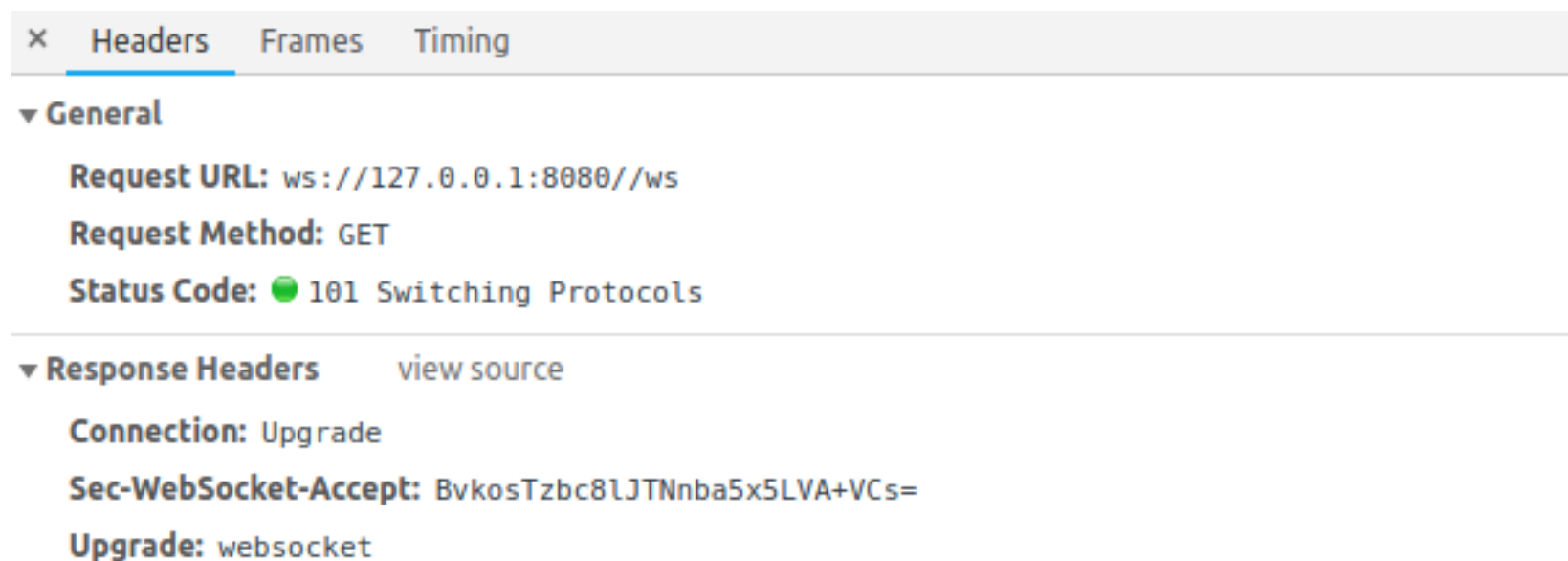
# УСТАНОВЛЕНИЕ СОЕДИНЕНИЯ

Для установления соединения браузер отправляет следующего вида запрос:



# УСТАНОВЛЕНИЕ СОЕДИНЕНИЯ

И ждёт ответ вида:



The screenshot displays the 'Headers' tab in a web browser's developer tools. The 'General' section is expanded, showing the following details:

- Request URL:** `ws://127.0.0.1:8080//ws`
- Request Method:** `GET`
- Status Code:** `101 Switching Protocols` (indicated by a green circle icon)

Below the 'General' section, the 'Response Headers' section is also expanded, showing:

- Connection:** `Upgrade`
- Sec-WebSocket-Accept:** `BvkosTzbc8lJTNnba5x5LVA+VCs=`
- Upgrade:** `websocket`

A 'view source' link is visible next to the 'Response Headers' section header.

# СЕРВЕРНАЯ ЧАСТЬ

Опять-таки, сервера, поддерживающего веб-сокеты, у нас нет, поэтому придётся его написать (в том же проекте):

```
npm install ws
```

```
1  const WS = require('ws');
2
3  // Предыдущий код
4
5  const port = process.env.PORT || 7070;
6  const server = http.createServer(app.callback());
7  const wsServer = new WS.Server({ server });
8
9  wsServer.on('connection', (ws, req) => {
10     const errorCallback = (err) => {
11         if (err) {
12             // TODO: handle error
13         }
14     }
15
16     ws.on('message', msg => {
17         console.log('msg');
18         ws.send('response', errorCallback);
19     });
20
21     ws.send('welcome', errorCallback);
22 });
23
24 server.listen(port);
```

# ПРОСМОТР ДАННЫХ

× Headers Frames Cookies Timing		
🔍 All ▼ Enter regex, for example: (web)?socket		
Data	Length	Time
↓ welcome	7	00:57:48.469

# ПЕРЕПОДКЛЮЧЕНИЕ

Ключевая вещь — при закрытии подключения браузер не будет автоматически переподключаться. Теперь эта задача лежит на нас.

Так, если мы «убьём» сервер, подключение закроется с кодом 1006 и уже мы сами будем решать, когда переподключаться и с каким интервалом.



# ОТПРАВКА ДАННЫХ

Зато в части отправки данных — полная свобода:

```
1 void send(USVString data);  
2 void send(Blob data);  
3 void send(ArrayBuffer data);  
4 void send(ArrayBufferView data);
```

Примечание\*: в `EventSource` желательно передавать только текстовые данные.

# РАССЫЛКА СООБЩЕНИЙ

Для нашего чата необходимо рассылать сообщения всем подключенным клиентам (серверная часть):

```
1 | Array.from(wsServer.clients)
2 |   .filter(o => o.readyState === WS.OPEN)
3 |   .forEach(o => o.send('some message'));
```

Ну а дальше получатели обрабатывают данное сообщение.



# ИТОГИ

Сегодня мы разобрали два новых API:

- SSE
- WebSockets

А также научились реализовывать серверную часть с использованием коа, http-event-stream, ws.



Спасибо за внимание!

Время задавать вопросы

**АЛЕКСЕЙ СУДНИЧНИКОВ**



[@avsudnichnikov](https://www.instagram.com/avsudnichnikov)