

# Echtzeit Kommunikation

TH Köln - Webtechnologien - Joshua Gawenda

# Agenda

1. 5 min Warum Echtzeit Kommunikation?
2. 5 min Stimmungsbild
3. 10 min Kapitel 1: Short Polling & Long Polling
4. 15 min Kapitel 2: Server-Sent Events
5. 25 min Aufgabe: Benachrichtigungen
6. 10 min Pause
7. 15 min Kapitel 3: WebSockets
8. 25 min Aufgabe: Live-Chat
9. 10 min Andere Technologien
10. 5 min Zusammenfassung



## Info

Fragen gerne immer reinrufen, das hier ist kein Monolog.

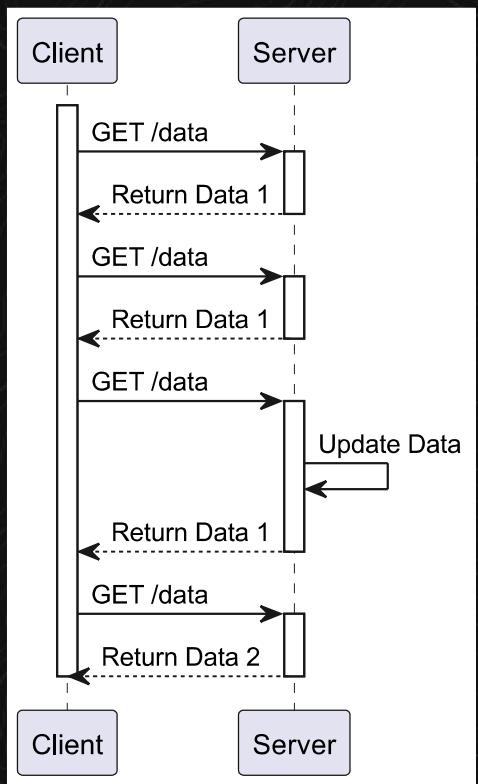
## **Warum Echtzeit Kommunikation?**

- Datenaustausch von 2+ Parteien
- Kommunikation mit absehbarer Latenz
- Reaktion auf Erstellung / Aktualisierung von Resources
- Anwendungsbereiche
  - Chat / Anrufe
  - Benachrichtigungen
  - Multiplayer Spiele
  - Navigations Apps
  - Vieles mehr
- Verbesserte UX
- Erhöhte Produktivität

# Polling

- Reguläre Abfrage einer Ressource
- Client → Server
- Server reagiert nur
- Nur Near-Realtime

## **Short Polling**



# Short Polling

- Server sendet immer Antwort
- Client hat einen festen Interval
- Datenänderung muss beim Client gecheckt werden
- Anwendung
  - Fremder Server
  - 100% Echtzeit nicht relevant
  - Bandbreite kein Problem

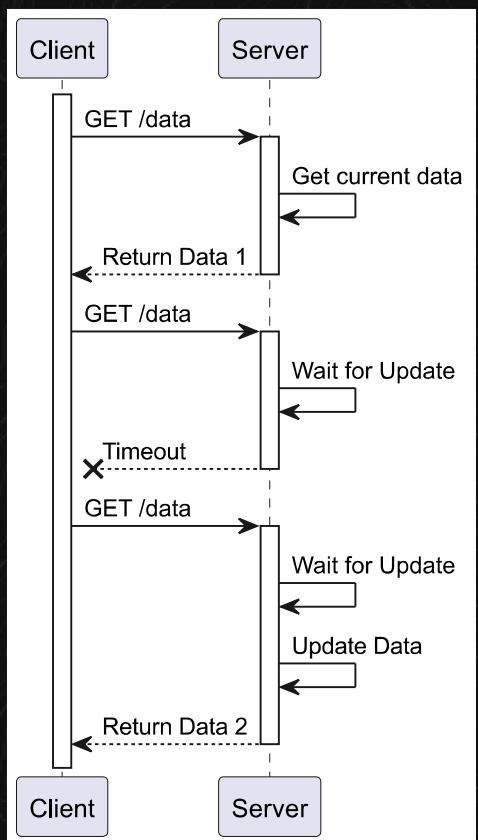
Pro	Contra
Immer Implementierbar	Hohe Netzwerkbelastung
Einfache Implementation	Interval bestimmt die "Echtzeit"
Rein Client-side	Schlecht Skalierbar
	Potentielles Rate-limiting
	Abfrage auch ohne Änderung

# Short Polling

client.js

```
1 function shortPoll() {
2     fetch('/api/data')
3         .then(response => {
4             // Process the received response
5         })
6
7     // Poll every 5 seconds
8     setTimeout(shortPoll, 5000);
9 }
10
11 // Start Short Polling
12 shortPoll();
```

# Long Polling



## Long Polling

- Auch regelmäßige Anfrage an Ressource
- Erst Antwort wenn neue Daten da sind
- In der Regel neue Anfrage nach Antwort
- Server Anpassung
- Anwendung
  - Eigener Server
  - Echtzeit ist relevanter
  - Kurze Entwicklungszeit

### Pro

Näher an Echtzeit als Short Polling	Nur eine Ressource pro Request
Reduziert Netzwerkbelastung	Nicht geeignet für "schnelle" Daten
Simple auf Server zu implementieren	Stateverlust durch mehrere Requests

### Contra

# Long Polling

client.js

```
1 function longPolling() {
2   fetch('https://api.example.com/updates')
3     .then(response => {
4       longPolling();
5       // do something with response
6     });
7 }
8
9 longPolling();
```

# Long Polling

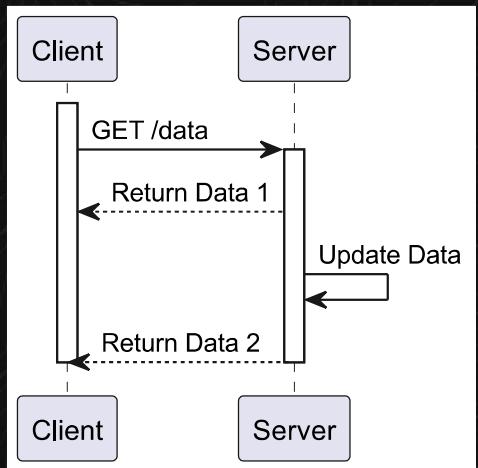
server.js

```
1 function handleLongPollingRequest(req, res) {
2   // wait for new data
3   const unsubscribe = addEventListener('newData', data => {
4     res.json(data);
5   });
6
7   // cleanup state when connection is closed
8   req.on('close', () => {
9     unsubscribe();
10  });
11 }
```

## **Server Sent Events (SSE)**

- 2006 eingeführt in "WHATWG Web Applications 1.0"
- Ursprünglich Server-sent DOM events
- Seit ca. 2010 in meisten Browsern
- HTTP/1.1 -> Limitierte Anzahl an Connections
- HTTP/2 (2015) -> Connection Multiplexing

## Server Sent Events (SSE)



# Server Sent Events (SSE)

- Regulärer HTTP-Request
- Client schickt 1 Request
- Server antwortet X mal
- Unidirektional (Server → Client)
- Text basierend
- 1 kontinuierlicher Stream
- Anwendung
  - Eigener Server
  - Echtzeit ist wichtig
  - Kurze Entwicklungszeit
  - Bidirectionale Kommunikation nicht gefordert
  - Only HTTP allowed

Pro	Contra
Echtzeit Kommunikation	Nur Unidirektional
Reduziert Netzwerkbela	Staltung an HTTP Connection auf HTTP/1.1
Eine einzige Connection	Limitiert Verfügbar in älteren Browsern
Einfach zu Entwickeln	

## Server Sent Events (SSE) Message Pattern

- Felder separiert durch 1 \n
- Messages separiert durch 2 \n
- Erstes Leerzeichen wird ignoriert

```
1 : Comment (can be used for keep alive)
2 retry: 10000
3 id: 1234
4 event: custom-event
5 data:some data here
6 data: Data in a new line
7 \n
```

## **Server Sent Events (SSE) Pitfalls**

- Compression könnte messages buffern
- Reverse Proxy braucht extra config
- Unbrauchbar bei HTTP/1.1

# Server Sent Events (SSE) Client

- Automatischer reconnect
- Message parsing
- Chrome Devtools
- Authentication nicht über header
- Nur im Browser
- Kann nur GET schicken

client.js

```
1 const eventSource = new EventSource('/events');
2
3 // Listen for messages without event type
4 eventSource.onmessage = (event) => {
5   console.log(event.data);
6 };
7 // Listen for messages with event type
8 eventSource.addEventListener('custom-event', (event) => {
9   console.log(event.data);
10});
```

# Server Sent Events (SSE) Client (fetch)

client-with-fetch.js

```
1 fetch("/events").then(async res => {
2     if(!res.body) return console.error('No body')
3     if(!res.headers.get('content-type').includes('text/event-stream')) return console.error('No SSE')
4     const reader = res.body.getReader()
5     const decoder = new TextDecoder()
6     let buffer = ''
7
8     while (true) {
9         const { done, value } = await reader.read()
10        if (done) break
11        buffer += decoder.decode(value)
12
13        if(!buffer.includes('\n\n')) continue
14
15        const [messages, restBuffer] = parseMessages(buffer)
16        buffer = restBuffer
17        // do something with messages
18    }
19 })
```

# Server Sent Events (SSE) Server

server.js

```
1 async function handleSSERequest(req, res) {
2   res.writeHead(200, {
3     'Content-Type': 'text/event-stream', // Tell Client to expect SSE
4     'Cache-Control': 'no-cache', // Disable caching
5     'Connection': 'keep-alive' // Tell Client to keep connection open
6   });
7
8   // send initial data
9   const currentData = await getData();
10  res.write(`event:initial\n`)
11  res.write(`data:${JSON.stringify(currentData)}\n`);
12  res.write('\n\n');
13
14  // wait for new data
15  const unsubscribe = addEventListener('newData', data => {
16    res.write('event:change\n');
17    res.write(`data:${JSON.stringify(data)}\n`);
18    res.write('\n\n');
19  });
20
21  // cleanup state when client closes connection
22  req.on('close', () => {
23    unsubscribe();
24  });
25 }
```

# Aufgabe - SSE Notifications

## 1. Client (client/index.js)

Nutze die **EventSource** Browser API um von dem Endpunkt "http://localhost:3000/sse" notification events zu erhalten. Lies folgende Events aus und zeige die Daten im Frontend an:

1. **initial**: Hier sind die bisherigen Daten als JSON encoded. Zeige diese in der Tabelle an.
2. **notification**: Eine neue Notification wurde gesendet, füge diese der Tabelle hinzu und flash die Nachricht.
3. **BONUS notification-update**: Es wurde vorher eine Notification gesendet die einen progress anzeigt, update diese mit den neuen informationen.

Das UI hat folgende methoden verfügbar

- **resetTable**: Löscht alle Notifications im Table.
- **addToTable**: Fügt eine Notification zum Table hinzu.
- **resetTable**: Löscht alle Notifications im Table.
- **flashNotification**: Flashed eine notification in einem toast.
- **updateFlashNotification**: Updated eine notification in einem toast.

## 2. Server (server/sse-route.js)

Ändere den Endpunkt im Frontend auf "http://localhost:4000/sse" und implementiere einen simplen Server Sent Events Endpunkt welcher:

1. Initial eine Liste an Notifications als JSON encoded sendet.
2. In einem random Interval eine neue Notification als notification event sendet.
3. **BONUS** Sende eine Progress Nachricht.

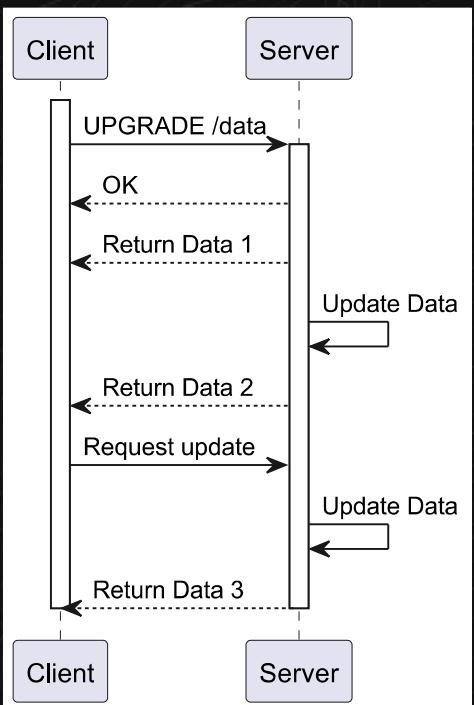
Eine Notification hat folgende Felder:

- **id**: Einzigartige Zahl oder String.
- **title**: Zusammenfassung der Nachricht.
- **message**: Die gesamte Nachricht.
- **time**: Der string timestamp der Nachricht.
- **type** (optional): Kann "progress" sein um eine Progress Nachricht anzugeben.
- **progress** (optional): Prozentualer Fortschritt einer Progress Nachricht.

# WebSockets

- Eigenes Protocol (ws:// | wss://)
- 2008 vorgestellt in HTML5 Spezifikation
- Seit ca. 2010 in meisten Browsern
- Seit 2024 nativ in NodeJS (v22)

# WebSockets



# WebSockets

- Bidirektionale Kommunikation
- Client schickt 1 Request + weitere Events
- Server antwortet X mal
- Keine Header oder Cookies\*
- 1 eigene TCP Connection
- Anwendung
  - Eigener Server
  - Echtzeit ist wichtig
  - Bidirectionale Kommunikation erforderlich
  - HTTP Overhead ist nicht erwünscht

## Pro

Echtzeit Kommunikation

Geringere Latency für Client -> Server

Eine einzige Connection

Bidirektional

## Contra

Anderes Protocol

Nicht überall verfügbar (bsp. Firewall oder Proxy)

Limitiert Verfügbar in älteren Browsern

Events nicht default

## **WebSockets Pitfalls**

- HTTP Header oder Cookie Auth nur bei initialem UPGRADE Request möglich
- HTTP Spezifische Logik kann nicht wiederverwendet werden (Middlewares, Context, etc)
- Socket.IO ist extrem beliebt aber nur innerhalb von Socket.IO nutzbar

# WebSockets Client

client.js

```
1 // Open new WebSocket connection
2 const ws = new WebSocket('ws://localhost:8080');
3
4 // Event Listener for when the connection is opened
5 ws.onopen = () => {
6     console.log('Connected to WebSocket server');
7 };
8
9 // Event Listener for when the connection is closed
10 ws.onmessage = (event) => {
11     console.log('Message from server:', event.data);
12 };
13
14 // Event Listener for when the connection is closed
15 ws.onclose = () => {
16     console.log('Disconnected from WebSocket server');
17 };
18
19 // Send a message to the server
20 function sendMessage() {
21     ws.send('Hello, server!');
22 }
```

# WebSockets Server

- Auch mit default **http** module möglich (UPGRADE handler)

server.js

```
1 // Using the 'ws' package, this could also be done with socket.io
2 import {WebSocketServer} from 'ws';
3 // Create a new WebSocket server
4 const server = new WebSocketServer({ port: 8080 });
5
6 // Event Listener for when the server is up and running
7 server.on('listening', () => {
8     console.log('Server is listening on port 8080')
9 });
10
11 // Event Listener for when a client connects
12 server.on('connection', (ws) => {
13     console.log('Client connected');
14
15     // Event Listener for when a client sends a message
16     ws.on('message', (message) => {
17         console.log('Received:', message);
18     });
19
20     // Event Listener for when a client disconnects
21     ws.on('close', () => {
22         console.log('Client disconnected');
23     });
24
25     // Send a message to the client
26     ws.send(`Hello, welcome to the WebSocket server!`);
27 });


```

# Aufgabe - WebSocket Chat

## 1. Client (client/index.js)

Nutze die **WebSocket** Browser API um von der URL "ws://localhost:8080" chat events zu erhalten und zu senden.  
Lies folgende Events aus und zeige die Daten im Frontend an und sende Chats an den Server:

Der Server initiiert events in der Kommunikation, von daher sind alle gesendeten und empfangenen Daten im Format `{ "event": "some event", "data": ... }`. Hier alle events:

1. **chat.initial**: Hier sind die bisherigen Nachrichten als JSON encoded. Zeige diese im Frontend an.
2. **chat.message**: Eine neue Nachricht wurde gesendet, füge diese dem Frontend hinzu.
3. **chat**: Sende eine neue Nachricht an den Server mit dem **username** und der **message**.

Das UI hat folgende methoden verfügbar

- **setFormDisabledState**: Enabled oder Disabled die Felder im Chat form.
- **setConnectionStatus**: Setzt den Verbindungsstatus im UI.
- **appendMessage**: Fügt eine Nachricht unten an den Rest.
- **setOnSubmit**: Setzt die Funktion die aufgerufen wird wenn das Form gesendet wird.

## 2. Server (server/index.js)

Ändere den Endpunkt im Frontend auf "ws://localhost:8080" und implementiere einen simplen WebSocketServer welcher:

1. Initial eine Liste an Chats als JSON encoded sendet.
2. Gesendete Chats empfängt, speichert und an alle Clients weiterleitet.

Eine Chat Message hat folgende Felder:

- **username**: Nutzer welche diese gesendet hat.
- **message**: Die enthaltene Nachricht.
- **timestamp**: Ein timestamp, welcher vom Server gesetzt werden soll.

Es gibt auch eine SSE version in den **index.sse.js** Files.

# WebRTC

- "New and improved WebSockets"
- Multiple Streams over one HTTP/3 Connection
- Gibt auch HTTP/2 Support (geringere Performance)
- 3 Transports:
  - Unidirection Stream (Server -> Client, Client -> Server)
  - Bidirectional Stream
  - Datagrams ("unreliable")
- Use Cases:
  - Low Latency Games
  - Out-of-order Media Dateien
  - Server Push Benachrichtigungen
- Working Draft -> Neuer Standard entwickelt von IETF
- Media Over QUIC (moq) in Arbeit für Mediastreams
- Bis auf Safari Support seit 2022
- Socket.IO supports WebTransport

# WebTransport

- Peer to Peer Kommunikation
- Audio, Video und Data Channels
- Protokolle:
  - Interactive Connectivity Establishment (ICE) für NAT Traversal
  - Session Traverse Utilities for NAT (STUN) für IP Adressen
  - Network Address Translation (NAT) für öffentliche IP Adressen
  - Traversing NATs and Firewalls (TURN) für Daten (Umgeht Symmetric NAT)
  - Session Description Protocol (SDP) für Media Streams
- Signaling Server für Verbindungsaufbau
- Für die meisten Conference Calls und Videokonferenzen genutzt
- Use Cases:
  - Video Calls
  - File Sharing
  - Screen Sharing
  - Live Streaming

# Zusammenfassung

- Viele Technologien für Echtzeitkommunikation
- Unterschiedliche Directions (Uni-, Bi-)
- Verschiedene Technologien für verschiedene Anwendungsfälle

