Verteilte Systeme

HTTP 1.1 – Teil C

by

Dr. Günter Kolousek

Client-seitiges Upgrade

1. Client

```
GET /hello.txt HTTP/1.1
Host: www.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of
  HTTP/2 SETTINGS payload>
```

2. Server

```
HTTP/1.1 101 Switching Protocols
Connection: upgrade
Upgrade: h2c
→ weiter mit HTTP/2
```

Server-seitiges Upgrade

1. Client

```
GET / HTTP/1.1
...
```

2. Server

```
HTTP/1.1 426 Upgrade Required
Upgrade: TLS/1.0, HTTP/1.1
Connection: upgrade
→ TLS Handshake...
```

Authentifizierung

- HTTP Basic Authentication
- HTTP Bearer Authentication
- HTTP Digest Authentication
- Formularbasierte Authentifizierung
  - basierend auf Cookies
  - basierend auf einem Token (siehe Bearer Authentication)
- HTTPS-Authentifizierung

Base64

- Kodierung
  - um binäre Daten über ASCII Kanal zu übertragen
- 3×8 Bits $\rightarrow 4 \times 6$ Bits
- $0 \rightarrow$ A,…,$26 \rightarrow$ a,…,$52 \rightarrow$ 0,…,$62 \rightarrow$ +, $63 \rightarrow$ /
  - padding: =
- Varianten
  - base64url: − anstatt +, _ anstatt /
    - zur Verwendung in Dateinamen und URLs
  - Radix-64: wie Base64 jedoch mit CRC-24 Prüfsumme am Ende
    - CRC … Cyclic Redundancy Check
  - weitere Abarten/Varianten…

Base64 – 2

source ASCII (if <128)	M								a								n							
source octets	77 (0x4d)								97 (0x61)								110 (0x6e)							
Bit pattern	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Index	19								22				5				46							
Base64-encoded	T								W				F				u							
encoded octets	84 (0x54)								87 (0x57)				70 (0x46)				117 (0x75)							

https://vikashazrati.files.wordpress.com/2008/07/base64.png

HTTP Basic Authentication

- Passwort mittels Base64 kodiert!
- Prinzip
  1. Browser

     ```
     GET /secure/index.html HTTP/1.0
     ```
  2. Server

     ```
     HTTP/1.0 401 Unauthorized
     ...
     WWW-Authenticate: Basic realm=”secureapp”
     ```
  3. Browser $\rightarrow$ Dialogbox

     ```
     GET /secure/index.html HTTP/1.0
     Authorization: Basic bWF4Om11c3Rlcg==
     ```

HTTP Basic Authentication

- Passwort mittels Base64 kodiert!
- Prinzip
  1. Browser

     ```
     GET /secure/index.html HTTP/1.0
     ```

  2. Server

     ```
     HTTP/1.0 401 Unauthorized
     ...
     WWW-Authenticate: Basic realm=”secureapp”
     ```

  3. Browser $\rightarrow$ Dialogbox

     ```
     GET /secure/index.html HTTP/1.0
     Authorization: Basic bWF4Om11c3Rlcg==
     ```

     ```
     import base64
     base64.decodestring(b”bWF4Om11c3Rlcg==”)
     # -> b'max:muster'
     ```

HTTP Bearer Authentication

- wie HTTP Basic Authentication
- aber: `WWW-Authenticate: Bearer realm=<realm>`
- Übermittlung des Tokens bei jedem Zugriff mittels
  - `Authorization: Bearer <token>`
  - über ein Formular (theoretisch)
  - Query-Parameter (theoretisch)

  aber auf keinem Fall in Cookies!

Passwörter

- Server speichert Passwort in DB
  - Client sendet Passwort und Server empfängt Passwort und vergleicht mit DB

Passwörter

- Server speichert Passwort in DB
  - Client sendet Passwort und Server empfängt Passwort und vergleicht mit DB
  - Client sendet Hashwert des Passwortes und Server berechnet Hashwert des in DB gespeicherten Passwortes

Passwörter

- Server speichert Passwort in DB
  - Client sendet Passwort und Server empfängt Passwort und vergleicht mit DB
  - Client sendet Hashwert des Passwortes und Server berechnet Hashwert des in DB gespeicherten Passwortes
- Server speichert Hashwert in DB
  - Client sendet Hashwert des Passwortes und Server vergleicht mit DB

Passwörter

- Server speichert Passwort in DB
  - Client sendet Passwort und Server empfängt Passwort und vergleicht mit DB
  - Client sendet Hashwert des Passwortes und Server berechnet Hashwert des in DB gespeicherten Passwortes
- Server speichert Hashwert in DB
  - Client sendet Hashwert des Passwortes und Server vergleicht mit DB
- Weitere Möglichkeiten mit $\rightarrow$ *Salt* und *Pepper*…

HTTP Digest Authentication

- ähnlich wie Basic Auth
- Prinzip
  1. Browser sendet Request
  2. Server sendet “nonce” (“Platzhalter”)
  3. Browser sendet Hash (MD5!) von
     - Benutzername
     - Passwort
     - realm
     - URI
     - Methode
     - nonce

HTTP Digest Authentication – 2

- anfällig für MITM Angriffe!

HTTP Digest Authentication – 2

- anfällig für MITM Angriffe!
  - $\rightarrow$ MITM verwendet HTTP Basic Authentication

HTTP Digest Authentication – 2

- anfällig für MITM Angriffe!
  - $\rightarrow$ MITM verwendet HTTP Basic Authentication
  - $\rightarrow$ MITM verwendet *rainbow table*
    Datenstruktur, um die ursprüngliche Zeichenfolge für einen Hashwert (Hashfunktion ohne Salt) zu ermitteln

HTTP Digest Authentication – 2

- anfällig für MITM Angriffe!
  - $\rightarrow$ MITM verwendet HTTP Basic Authentication
  - $\rightarrow$ MITM verwendet *rainbow table*
    Datenstruktur, um die ursprüngliche Zeichenfolge für einen Hashwert (Hashfunktion ohne Salt) zu ermitteln
  - $\rightarrow$ MITM versucht *chosen-plaintext attacks*
    Aus Geheimtext für gewählten Klartext wird versucht den Schlüssel zu ermitteln

HTTP Digest Authentication – 2

- anfällig für MITM Angriffe!
  - $\rightarrow$ MITM verwendet HTTP Basic Authentication
  - $\rightarrow$ MITM verwendet *rainbow table*
    Datenstruktur, um die ursprüngliche Zeichenfolge für einen Hashwert (Hashfunktion ohne Salt) zu ermitteln
  - $\rightarrow$ MITM versucht *chosen-plaintext attacks*
    Aus Geheimtext für gewählten Klartext wird versucht den Schlüssel zu ermitteln
- Erweiterungen in RFC 2617 um
  - client-nonce

HTTP Digest Authentication – 2

- anfällig für MITM Angriffe!
  - $\rightarrow$ MITM verwendet HTTP Basic Authentication
  - $\rightarrow$ MITM verwendet *rainbow table*
    Datenstruktur, um die ursprüngliche Zeichenfolge für einen Hashwert (Hashfunktion ohne Salt) zu ermitteln
  - $\rightarrow$ MITM versucht *chosen-plaintext attacks*
    Aus Geheimtext für gewählten Klartext wird versucht den Schlüssel zu ermitteln
- Erweiterungen in RFC 2617 um
  - client-nonce
    - $\rightarrow$ wg. *chosen-plaintext attacks* und *rainbow tables*
  - request counter

HTTP Digest Authentication – 2

- anfällig für MITM Angriffe!
  - $\rightarrow$ MITM verwendet HTTP Basic Authentication
  - $\rightarrow$ MITM verwendet *rainbow table*
    Datenstruktur, um die ursprüngliche Zeichenfolge für einen Hashwert (Hashfunktion ohne Salt) zu ermitteln
  - $\rightarrow$ MITM versucht *chosen-plaintext attacks*
    Aus Geheimtext für gewählten Klartext wird versucht den Schlüssel zu ermitteln
- Erweiterungen in RFC 2617 um
  - client-nonce
    - $\rightarrow$ wg. *chosen-plaintext attacks* und *rainbow tables*
  - request counter
    - $\rightarrow$ wg. *replay attacks*

Einschub: Wörterbuchangriff

- Hashwert zu einem Passwort immer derselbe
- Vorberechnung möglich: Wörterbuchangriff
  - $\rightarrow$ Rainbow Tables
- Salt
  - Server erzeugt je Passwort zufällige Zeichenfolge und speichert diese (Salt)
  - Kombination mit Salt
  - Berechnung des Hashwertes
- Pepper
  - wie Salt, aber für alle Passwörter gleich
  - dafür wird dieser **nicht** in der Datenbank gespeichert sondern extern an einem sicheren Ort
  - $\rightarrow$ Auch wenn Angreifer Zugriff auf Datenbank erhält (z.B. mittels SQL-Injection) sind keine realistischen Angriffe auf die Passwörter möglich

Rainbow Table – Aufbau

- Hashfunktion H und Reduzierungsfunktion R
  - R … beliebige Funktion, die Hashwert in eine Klartextzeichenkette wandelt
- Annahme: 6 stellige Passwörter und Hashfunktion, die 32 Bithashwerte liefert
  - $\texttt{aaaaaa} \underset{H}{\rightarrow} \texttt{281DAF40} \underset{R}{\rightarrow} \texttt{sgfnyd} \underset{H}{\rightarrow} \texttt{920ECF10} \underset{R}{\rightarrow} \texttt{kiebgt}$
- Tabelle
  1. Wähle beliebige Anzahl an Anfangspasswörter
  2. Berechne von jedem eine fixe Kette der Länge k und speichere jeweils Anfang und Ende

Rainbow Table – Prinzip

- Vorgang
  1. Berechne für gegebenen Hashwert die Kette bis zum Ende
  2. Beginne am Anfang dieser Kette und verfolge diese bis nächster Hashwert erreicht wird
- Beispiel
  1. `920ECF10` $\underset{R}{\rightarrow}$ `kiebgt`
  2. `aaaaaa` $\underset{H}{\rightarrow}$ `281DAF40` $\underset{R}{\rightarrow}$ `sgfnyd` $\underset{H}{\rightarrow}$ `920ECF10`
     - d.h. Passwort ist `sgfnyd`!

Rainbow Table – Prinzip

- Vorgang
  1. Berechne für gegebenen Hashwert die Kette bis zum Ende
  2. Beginne am Anfang dieser Kette und verfolge diese bis nächster Hashwert erreicht wird
- Beispiel
  1. `920ECF10` $\underset{R}{\rightarrow}$ `kiebgt`
  2. `aaaaaa` $\underset{H}{\rightarrow}$ `281DAF40` $\underset{R}{\rightarrow}$ `sgfnyd` $\underset{H}{\rightarrow}$ `920ECF10`
     - d.h. Passwort ist `sgfnyd`!
     - wenn nicht enthalten, dann “falscher Alarm” $\rightarrow$ verwerfen und Kette bis max. zur Länge k weiterverfolgen. Wenn dann nicht enthalten, dann wurde das Passwort in keiner Kette generiert.

Tunneling Proxy

- Relais zwischen 2 Verbindungen
  - im Standard von HTTP/1.1: “Tunnel”
- Zweck
  - unveränderte Weitergabe
  - meist: TLS zwischen Client und Server via Proxy
    - $\rightarrow$ `CONNECT`

Reverse Proxy

- Serverseite
  - im Standard von HTTP/1.1: “Gateway”
- Zweck
  - Lastverteilung (load balancing)
    - ankommende Requests werden aufgeteilt
  - Caching
  - Sicherheit
    - Verschlüsselung: TLS (u.U. eigene HW)
    - Zusätzliche Schicht

(Forward) Proxy

- Clientseite
  - im Standard von HTTP/1.1: “Proxy”
- Zweck
  - Caching
  - Sicherheit
    - Verschlüsselung, Zugriffskontrolle
  - Authentifizierung
  - Loggen
  - Verändern der Nachricht (z.B. Bildformate)
    - speziell: Kompression
  - Filterung von Inhalten
  - Anonymisierung
- absolute URL!

proxy.pac – Beispiel

```
function FindProxyForURL(url, host) {
    if (shExpMatch(host, "*.htlwrn.ac.at")) {
        return "DIRECT";
    }

    if (isInNet(host, "10.0.0.0", "255.0.0.0")) {
        return "PROXY intprox.htlwrn.ac.at:8080";
    }

    return "PROXY proxy.htlwrn.ac.at:8080;";
}
```

Webanwendungen

- client-seitig
  - Interaktivität
  - JavaScript (TypeScript, Dart, CoffeeScript), WebAssembly
  - $\rightarrow$ SPA (single page application)
    - ein HTML Dokument, u.U. Inhalte dynamisch nachladen
    - Angular, Vue.js, Knockout.js,…
- server-seitig
  - Sicherheit
  - Rechenleistung
  - zentraler Datenbestand
  - PHP, Python, Ruby, Java, C#,…
    - CGI, FastCGI, SCGI, WSGI,…
    - Apache Modules, ISAPI
    - ASP.NET, Java Servlets (JSP), JEE, Node.js,…
    - RoR (Ruby on Rails), Django, Flask, Vaadin,…

CGI

- Common Gateway Interface
- Problematik: Funktionen am Server im Kontext von Webanwendungen ausführen
- Prinzip
  - starten eines Prozesses
  - Input: Umgebungsvariable, stdin, Kommandozeilenparameter
  - Output: stdout
- Nachteil: Latenz!
- Alternativen
  - FastCGI: 1 laufender Prozess!
  - SCGI (Simple CGI): wie FastSCGI, aber einfacher
  - WSGI (Web Server Gateway Interface): speziell für Python

HTTP – Interaktivität

- prinzipiell gering
  - $\rightarrow$ Request/Response
    - neuer Aufbau einer Seite!
  - daher auch keine “Echtzeitfähigkeit”
- Lösungsansätze
  - “Tool”: XMLHttpRequest (XHR) !
  - Polling
  - Long-Polling
  - Comet
  - Server-Sent Events !!
  - WebSockets !!!

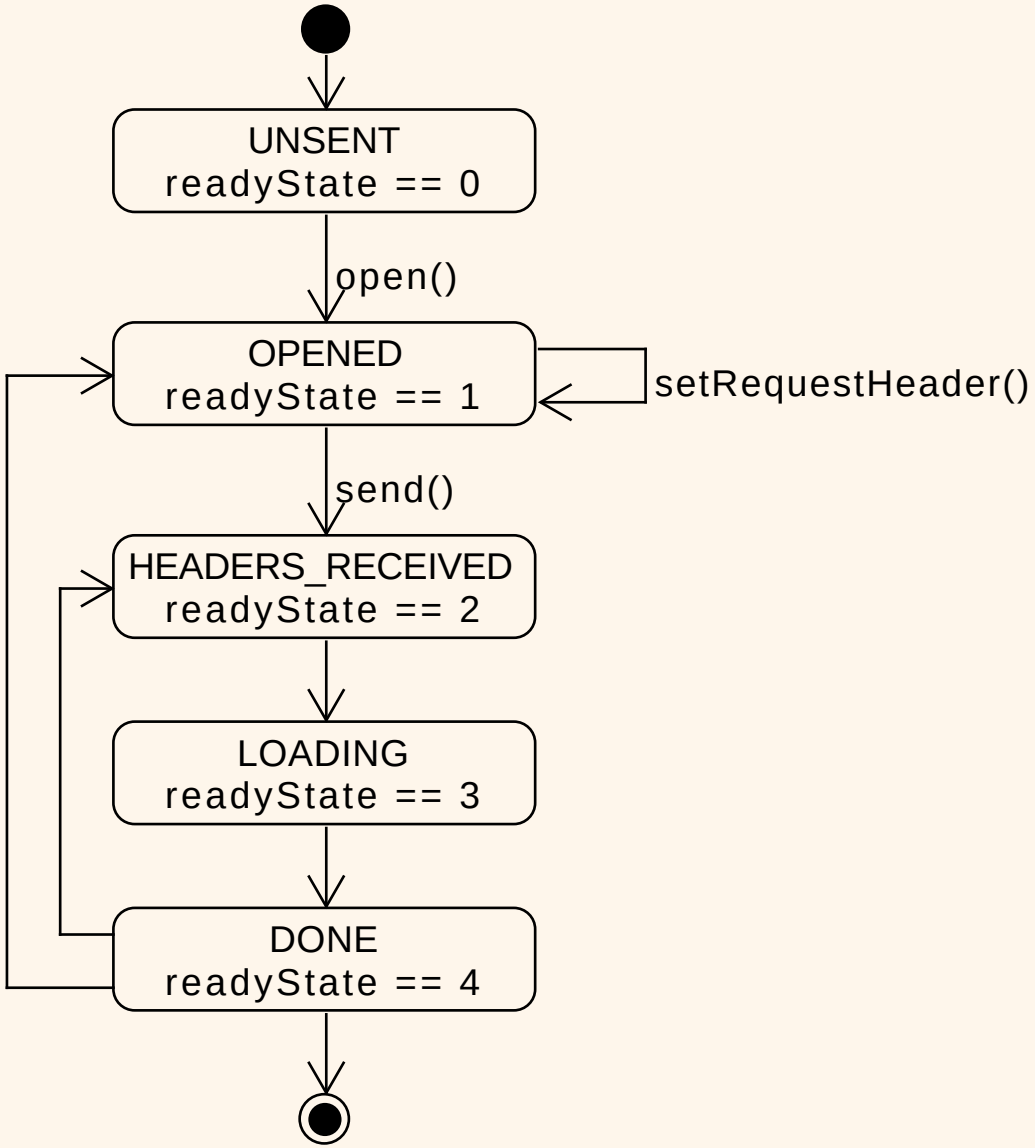
XHR

```
var xhr = new XMLHttpRequest();
// true -> async!
xhr.open("GET", "http://if.htlwrn.ac.at", true);
xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200)
        console.log(this.responseText);
    }
}
xhr.send();
```

→ Ajax (Asynchronous Javascript And XML)

Tipp: besser z.B. jQuery verwenden

XHR – Zustandsdiagramm



Polling

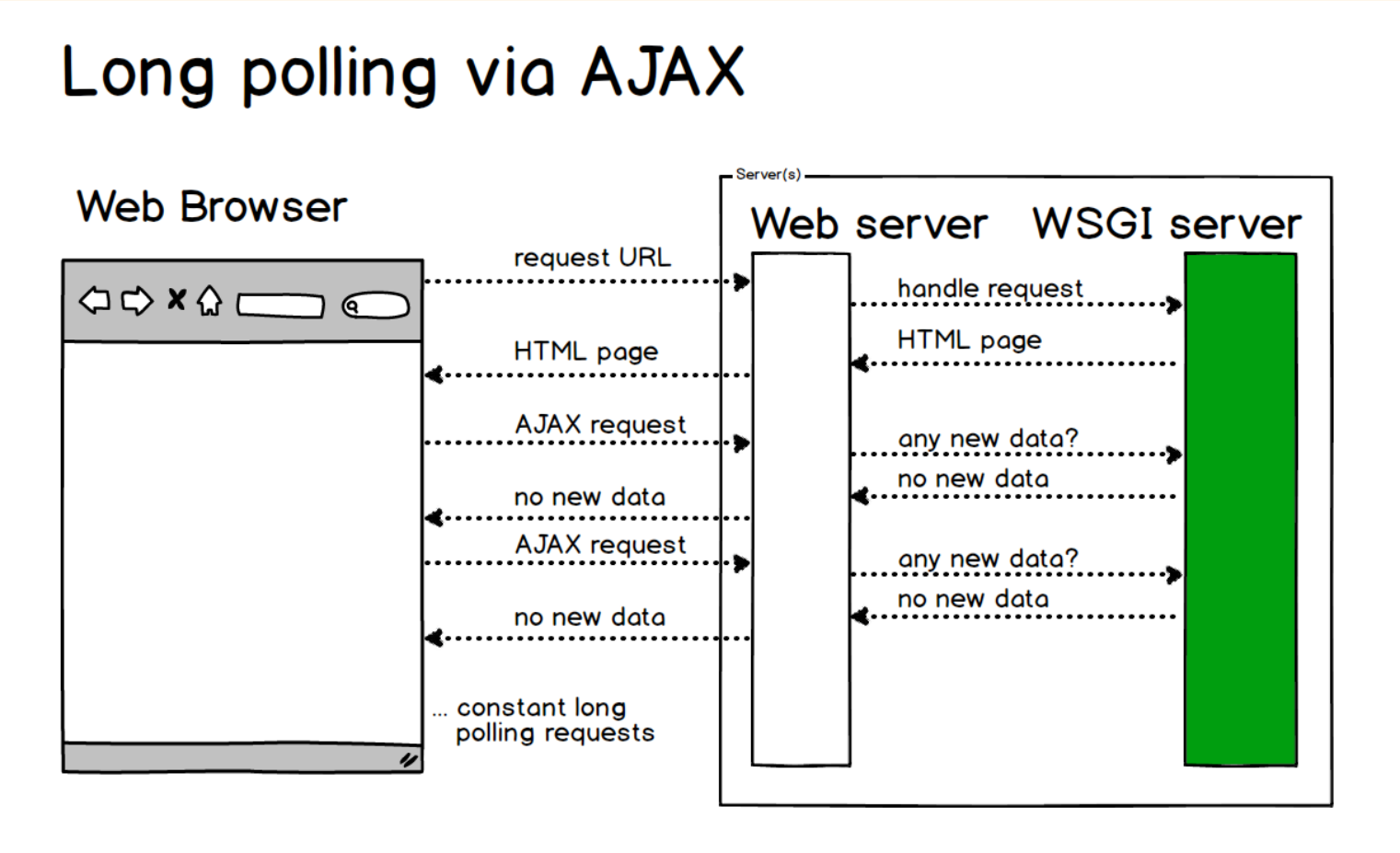
```
function poll() {
    xhr.send();
    clearTimeout(timeout_id);
    timeout_id = setTimeout("poll()", 2000);
}
timeout_id = setTimeout("poll()", 2000);
```

keine Daten $\rightarrow$ leere Antwort

Long-Polling

- wie Polling, aber:
- Antwort
  - sofort, wenn Daten vorhanden
  - nach Ablauf einer Zeitspanne, dann leere Antwort
  - Verbindung muss (natürlich) offen bleiben
- nach Antwort: umgehend neuer Request

Long-Polling – 2



Quelle:
https://www.fullstackpython.com/websockets.html

Comet

- ebenfalls Putzmittel…
- keine
  - einheitliche Definition → Oberbegriff
  - Standardardisierung
- wie Long-Polling, aber:
  - JSONP (JSON with Padding) anstatt XHR
  - `script` Element dynamisch erzeugen
    - und zu DOM hinzufügen
    - `script` hebelt SOP (Same Origin Policy) aus! → CORS (Cross-Origin Resource Sharing → *Websicherheit*)
- SOP → Folien *Websicherheit*
  - Skripte einer “Seite” dürfen nur auf Informationen dieser “Seite” zugreifen
  - wenn Domäne | Protokoll | Port unterschiedlich → kein Zugriff

<script> mit JSONP – 1

- `script` Element

```
<script type="text/javascript"
  src="http://if.htlwrn.ac.at/getjson?
      callback=parseResponse">
</script>
```

- Antwort

```
parseResponse({"name": "foo", "id": 4711});
```

<script> mit JSONP – 2

```
<script>
function parseResponse(data) {
    // process data
};

// a new script element
var elem = document.createElement('script');

elem.src = ”http://if.htlwrn.ac.at/getjson?” +
           ”callback=parseResponse”;

// add it to <head>
document.getElementsByTagName('head')[0].
  appendChild(elem);
</script>
```

→ besser z.B. jQuery verwenden!

Server-Sent Events (SSE)

- HTML5
- Nachrichten von Server zu Client
- definiert API
- definiert Protokoll
- Nicht: IE

SSE – API

```
var es = new EventSource("messages");
es.onmessage = function(event) {
    msg_div = document.getElementById("msg");
    msg_div.innerHTML += "<br/>" + event.data;
};
<body>
    <div id="msg"></div>
</body>
```

SSE – API – EventSource

`url` (readonly)

`CONNECTING` 0 (readonly, const)

`OPEN` 1 (readonly, const)

`CLOSED` 2 (readonly, const)

`readyState` (readonly)

`onopen` (function)

`onmessage` (function)

`onerror` (function)

void `close()`

SSE – API – Events

```
// if an event field is available (see later)
es.addEventListener("user_connect", function(e) {
  var new_item = document.createElement("li");

  // data: {"user": "maxi", "time": "2016-11-06"}
  var o = JSON.parse(e.data);
  new_item.innerHTML = "user " + o.user + " at "
                              + o.time;
  eventList.appendChild(new_item);
});
```

`message` ist der Default als Eventtyp.

SSE – Protokoll – Request

```
GET /messages HTTP/1.1
...
Accept: text/event-stream
Last-Event-ID: 6                # nicht im API!!!
Cache-Control: no-cache
```

SSE – Protokoll – Response

```
HTTP/1.1 200 OK
...
Content-Type: text/event-stream
Expires: Mon, 1 Jan 2001 00:00:00 GMT
Cache-Control: no-cache, no-store, max-age=0,
  must-revalidate
Pragma: no-cache
Connection: close

id: 7
data: {”key”: ”foo”, ”value”: 4711}

id: 8
data: {”key”: ”bar”, ”value”: 1503}

...
```

SSE – Protokoll – Response – 2

- Feldnamen

`id`	last event ID wird auf diesen Wert gesetzt
`data`	die Daten…
`event`	der Eventname, z.B. ”`user_connect`” → `addEventListener()`
`retry`	Zeit in ms wenn UA die Verbindung geschlossen hat und danach wieder öffnet

- Kommentar: Zeile beginnt mit `:`