Web-basierte Systeme

06: Browser Kommunikationsschnittstellen

Wintersemester 2022

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Browser Kommunikationsschnittstellen

Vorlesungsplan

2. November	Einführung und Darstellung von Webseiten (HTML und CSS)
9. November	Hypertext Transfer Protocol
16. November	Browser Schnittstellen
23. November	Kommunikationsschnittstellen im Browser
30. November	WebAssembly
7. Dezember	**Architektur moderner Browser**
14. Dezember	Clientseitige Architekturmuster
21. Dezember	Vorbereitung Papieranalyse
11. Januar	Papieranalyse
18. Januar	Serverseitige Implementierung von Web-basierten Systemen
25. Januar	Caching bzw. Lastverteilung durch Zwischenspeicher
1. Februar	Thema noch offen – wahrscheinlich WebRTC
8. Februar	Zusammenfassung und Ausblick

Browser Kommunikationsschnittstellen

Zielsetzung der Lerneinheit

- Verständnis über die Kommunikationsschnittstellen zwischen Browser und Server sowie die Vor- und Nachteile der jeweiligen Mechanismen

XMLHttpRequest

XMLHttpRequest

Entstehung und Motivation von XMLHttpRequest (XHR)

- Zu Anfang erforderte eine Aktualisierung einer Webanwendung das Laden einer neuen Seite
- Mittels XHR konnten nun direkt aus einem Skript Anfragen an den Server gestellt werden – die im Hintergrund laufen
  - Basis für die *Asynchronous JavaScript and XML (AJAX)*-Revolution
- XHRs wurden mit dem Internet Explorer 5 eingeführt und quasi durch *Zufall* mit XML verknüpft
- Mozilla und weitere Browserhersteller zogen recht schnell nach
- 2006 erste W3C Working Draft Spezifikation
- 2008 gab es einen weiteren Entwurf mit *XMLHttpRequest Level 2*
- 2011 wurden beide Versionen fusioniert und es gibt quasi nur noch eine XMLHttpRequest Spezifikation
  - (`https://xhr.spec.whatwg.org`)

XMLHttpRequest

Einbettung von XHR in den Browser

- XHR ist eine recht komfortable Kommunikationsschnittstelle und regelt viele Details im Browser
  - Verwalten der Anfrage, caching, Authentifizierung, etc.
  - Es werden aber auch Sicherheitsmechanismen etabliert bzw. genutzt
- Es können zwar zusätzliche HTTP Header einer Anfrage hinzugefügt werden (`setRequestHeader( )`) aber die Art der Header ist beschränkt:
  - Beispielsweise sind `Access-Control-*, Cookie, Origin` und `Refer` nicht möglich! (Der Browser verwirft diese einfach.)
  - Wichtig um sicherzustellen das der `Origin`-Header nicht verändert werden kann (*same-origin-policy*)
    - (Zwei URLs haben den gleichen *origin* falls Protokoll, Port und Host gleich sind.)
  - Zugriff auf Daten (cookies) von anderen Websites wäre möglich!

XMLHttpRequest

XMLHttpRequest Beispiel

```
function reqListener () {
  console.log(this.responseText);
}

var xhr = new XMLHttpRequest();
xhr.addEventListener("load", reqListener);
xhr.open("GET", "http://www.example.org/blub.txt");
xhr.send();
```

XMLHttpRequest

Unterstützte Datenformate

- `ArrayBuffer` - Binärdaten fester Länge
- `Blob` - Binärdaten (die im Browser nicht verändert werden)
- `Document` - HTML oder XML Daten
  - Bsp. für Nutzung: `elem.innerHTML = xhr.responseText;`
- `JSON`
  - Bsp. für Nutzung: `JSON.parse(xhr.responseText);`
- `Text`

Bestimmung des Datenformates

- Entweder durch den HTTP content-type der Antwort
- ...oder explizit durch die Anwendung
  (via `xhr.responseType = 'blob';`)

XMLHttpRequest

Ereignisse

- `readystatechange`: wenn sich der Status geändert hat
- `loadstart`: Anfrage wurde gestartet.
- `progress`: Daten werden gesendet oder empfangen.
- `load`: Anfrage wurde erfolgreich abgeschlossen
- `loadend`: Anfrage wurde beendet (egal ob erfolgreich/gescheitert)
- `abort` und `error`

Mögliche Verarbeitungszustände via `readyState`

0. **UNSENT**: `open( )` ist noch nicht aufgerufen worden
1. **OPENED**: `send( )` aufgerufen
2. **HEADERS_RECEIVED**: send() aufgerufen und headers verfügbar
3. **LOADING**: Download nicht abgeschlossen & Teilergebnis verfügbar
4. **DONE**: Operation ist abgeschlossen

XHR Beispiel für das Landen von Binärdaten

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('GET', '/images/some.jpg');
3  xhr.responseType = 'blob';
4
5  xhr.onload = function() {
6    if (this.status == 200) {
7      var img = document.createElement('img');
8      img.src = window.URL.createObjectURL(this.response);
9      img.onload = function() {
10         window.URL.revokeObjectURL(this.src);
11     }
12     document.body.appendChild(img);
13   }
14 };
15 xhr.send();
```

XMLHttpRequest

XHR Beispiel für Upload

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('POST','/upload');
3  xhr.onload = function() { ... };
4  xhr.send("text string");
```

- Up- und Download sind sehr ähnlich zueinander
- Es wird keine strom-orientierte Übertragung unterstützt, dies kann aber emuliert werden, indem mehrere Anfragen versendet werden

XHR Fortschrittsüberwachung

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('GET','/resource');
3  xhr.timeout = 5000;
4  ...
5  var onProgressHandler = function(event) {
6    if(event.lengthComputable) {
7      var progress = (event.loaded / event.total) * 100;
8      ...
9    }
10 }
11 xhr.addEventListener('progress', onProgressHandler);
12 xhr.send();
```

- Über den Fortschritt von Uploads kann man sich analog informieren: `xhr.upload.addEventListener('progress', onProgressHandler);`

XMLHttpRequest

Echtzeitbenachrichtigung

- Es sollen neue Meldungen angezeigt werden, sobald sie auf der Serverseite verfügbar sind
- Naive Lösung mit XHR - *polling*:

```
1  function checkUpdates(url) {
2    var xhr = new XMLHttpRequest();
3    xhr.open('GET', url);
4    xhr.onload = function() { ... };
5    xhr.send();
6  }
7  setInterval(function() { checkUpdates('/up') }, 60000);
```

- Ist das gut? Geht das besser?

XMLHttpRequest

Naives Polling

- Naives Polling ist kritisch zu sehen, da ein blankes Anfrage/Antwort-Paar auch ohne neue Nachrichten 800 Byte Datentransfer verursacht!
- Dies ist ineffizient und nicht besonders energiefreundlich für mobile Endgeräte
- Weithin wird die Zustellung von neuen Nachrichten um bis zu Polling-Intervall verzögert!

Gibt es dazu eine Alternative?

XMLHttpRequest

Verzögertes Polling oder auch *Comet*

- Liegt serverseitig keine neue Nachricht vor so wird die Antwort verzögert bis dies der Fall ist
- Client schickt nach Erhalt einer Antwort eine neue Anfrage los

```
1  function checkUpdates(url) {
2    var xhr = new XMLHttpRequest();
3    xhr.open('GET', url);
4    xhr.onload = function() {
5      ...
6      checkUpdates('/updates');
7    };
8    xhr.send();
9  }
10 checkUpdates('/updates');
```

- Ist das jetzt in jedem Fall eine gute Lösung?

Cross-Origin Resource Sharing (CORS)

- Manchmal soll ein Skript Ressourcen eines fremden Server abfragen
- Lösung bildet CORS

XMLHttpRequest

Cross-Origin Resource Sharing (CORS)

- Browser stellt durch Header erweiterte Anfrage an den Ursprungsserver dieser kann dann entscheiden

Anfrage mit fremden `Origin:` der durch den Browser kontrolliert wird

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
...
Origin: http://foo.example
```

Antwort: Uneingeschränkter Zugriff ist möglich (?!)

```
HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: *

[Data]
```

XMLHttpRequest

Cross-Origin Resource Sharing (CORS)

- Man unterscheidet zwischen einfachen bzw. *simple cross-origin requests* und erweiterten Anfragen
- Einfache Anfragen sind auf bestimmte HTTP Methoden begrenzt (GET, POST, HEAD) während erweiterten Anfragen für Cookies und eigene Header nötig sind
- In diesem Fall muss eine Voranfrage (*preflight request*) gestellt werden
- Konsequenz ist ein extra Nachrichtenaustausch – der aber zwischengespeichert werden kann

CORS Beispiel für Voranfrage

```
OPTIONS /resource.js HTTP/1.1
Host: thirdparty.com
Origin: http://example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: My-Custom-Header
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://example.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: My-Custom-Header
```

XMLHttpRequest

Zusammenfassung und Fazit

- XHR ermöglicht interaktive Webseiten ohne das Laden einer neuen Seite
- Sehr komfortable und einfach, da die Mechanismen des Browsers genutzt werden
- Austausch von Echtzeitnachrichten möglich aber nicht optimal besser bspw. Server-Sent Events (SSE)
- Stromorientierter Nachrichtenaustausch ist nicht gut unterstützt!

Server-Sent Events

Server-Sent Events

Motivation und Eigenschaften

- Server-Sent Events (SSE) setzt sich aus zwei Komponenten zusammen:
  - der EventSource-Schnittstelle des Browsers welches die Server-initiierte Zustellung von *push notifications* als DOM Ereignisse realisiert
  - und einem stromgewahren Datenformat

- SSE stellt eine stromorientierte Nachrichtenschnittstelle bereit

Einfaches Beispiel

```
1  var source = new EventSource("news"); //relative URL
2
3  source.onopen = function () { ... };
4  source.onerror = function () { ... };
5
6  // Register handler for event of type "fun"
7  source.addEventListener("fun", function (event) {
8    processFun(event.data);
9  });
10
11 source.onmessage = function (event) {
12   log_message(event.id, event.data);
13   if (event.id == "CLOSE") {
14     source.close();
15   }
16 }
```

Server-Sent Events

Eigenschaften der zugrunde liegenden Infrastruktur

- Etablierung der Verbindung
- Empfangene Daten werden inkrementell aufbereitet (z.B. identifizieren von Nachrichtengrenzen usw.)
- Auslösen entsprechender Ereignisse
- Zusätzlich wird die Wiederherstellung von abgebrochenen Verbindungen unterstützt
  - Unterstützung für zuverlässige Übertragung mittels `Nachrichten-IDs` ein

Server-Sent Events

Event Stream Protocol und dessen Abbildung auf HTTP

- Browser versendet reguläres `HTTP GET` mit Anfrage nach SSE Ereignissen (`text/event-stream`)

```
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream
```

Server-Sent Events

Event Stream Protocol und dessen Abbildung auf HTTP

- Server antwortet mit einem Strom von Nachrichten (chunked)

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry: 15000

data: First message is a simple string.

data: {"message": "JSON payload"}

event: foo
data: Message of type "foo"

id: 42
event: bar
data: Multi-line message of
data: type "bar" and id "42"
```

Server-Sent Events

Nachrichtenformat

- Server schlägt ein Intervall vor, nachdem versucht werden soll eine abgebrochene Verbindung zu reetablieren
- Es können einfache Text- bzw. JSON-basierte Nachrichten versendet werden
- Optional können Nachrichten mit einem *Typ* versehen werden
- Nachrichten können *IDs* erhalten und mehrere Zeilen in Anspruch nehmen

Server-Sent Events

Erneute Versendung bei Verbindungsabbruch

- Durch Hinzufügen von IDs kann der Browser Buch führen, welches Ereignis er zuletzt vor einem Verbindungsabbruch zugestellt hat
- Diese Information wird bei einem erneuten Verbindungsaufbau durch einenden `Last-Event-ID` Header dem Server mitgeteilt
- Verbindungsabbruch und nun erneuter Aufbau ...

```
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream
Last-Event-ID: 42
```

- Antwort

```
HTTP/1.1 200 OK
Content-Type: text/event-stream
Connection: keep-alive
Transfer-Encoding: chunked

id: 43
data: bar foo
```

Zusammenfassung

- SSE ermöglicht einfaches vermitteln von Echtzeitinformationen
- Basis ist die verbindungs- und stromorientierte Kommunikation
- Da SSE auf HTTP aufsetzt ist auch Kompression (gzip) möglich
- Verbindungsabbrüche werden erkannt und behoben und verlorenen Nachrichten werden wiederholt übertragen
- SSE hat dennoch zwei Schwachstellen
  - Es bietet keine Unterstützung für eine stromorientierte Übertragung vom **Browser zum Server**
  - Lediglich eine UTF-8 basierte Übertragung ist vorgesehen – dies erschwert die effiziente Übertragung von Binärdaten
    - Entweder base64-Kodierung – was eher ineffizient ist
    - Alternativer Übertragungskanal und nur Signalisierung über SSE

WebSocket

WebSocket

Motivation und Eigenschaften

- Bidirektionale stormorientierte Verbindung zum Austausch von Text und Binärdaten
- Look-and-feel in etwa wie Unix-Sockets bietet aber zusätzliche Funktionen an:
  - Same-origin policy wird unterstützt
  - Kompatibel zum HTTP-Ökosystem
  - Effizientes Nachrichtenformat
  - Erweiterbar
    - Bspw. kann die Nutzung eines Unterprotokolls ausgehandelt werden
- WebSocket-Schnittstelle wird durch den W3C definiert
- Protokoll zum Datenaustausch durch den RFC 6455 festgelegt
- Erweiterbarkeit wird durch die HyBi Working Group (IETF) entwickelt

Einfaches Beispiel

```
1  var ws = new WebSocket('wss://example.com/socket');
2
3  ws.onerror = function (error) { ... }
4  ws.onclose = function () { ... }
5  ws.onopen  = function () { ws.send("Hello server!"); }
6
7  ws.onmessage = function(msg) {
8    if(msg.data instanceof Blob) {
9      processBlob(msg.data);
10   } else {
11     processText(msg.data);
12   }
13 }
```

- Nachrichten können als Text oder Binärdaten übertragen werden

WebSocket URL Schema

- WebSocket verfügt über zwei eigene Protokollkürzel
  - `ws` für Klartextkommunikation (bspw. `ws://example.com/socket`)
  - `wss` für verschlüsselte Kommunikation via TCP und TLS
- Warum gibt es ein eigenes Protokollkürzel?
  - De-facto werden WebSockets vornehmlich verwendet um Verbindungen zwischen Browser und Server zu etablieren aber im Prinzip ist die Verwendung auch außerhalb vorstellbar
  - Die HyBi Working Group hat deshalb ein eigenes URL-Schema eingeführt

Empfang und Verarbeitung von Daten

- Über WebSockets werden ganze Nachrichten verschickt/empfangen
  - Anwendungscode muss die Daten nicht zwischenspeichern, parsen oder Nachrichten wieder zusammenzusetzen
- Nachrichten können Text/Binärdaten (`Blob`) sein, auch abwechselnd
  - In Nachrichtenformat wird vor allem die Länge und der Typ kodiert

```
var ws = new WebSocket('wss://example.com/socket');
ws.binaryType = "arraybuffer";

ws.onmessage = function(msg) {
  if(msg.data instanceof ArrayBuffer) {
    processArrayBuffer(msg.data);
  } else {
    processText(msg.data);
  }
}
```

- Binärdaten (`Blob`) können automatisch in einen `ArrayBuffer` umgewandelt werden

Empfang und Verarbeitung von Binärdaten

- `ArrayBuffer` wird sicherlich im Arbeitsspeicher gehalten
  - Einfaches Abbilden auf Datenstrukturen möglich
  - `var usernameView = new Uint8Array(buffer, 0, 16);`
- Blob wird eher als Datei abgelegt und es wird davon ausgegangen, dass die Daten unveränderlich sind

Versenden und Verarbeitung von Daten

```
var ws = new WebSocket('wss://example.com/socket');

ws.onopen = function () {
  socket.send("Hello server!");
  socket.send(JSON.stringify({'msg': 'payload'}));

  var intview = new Uint32Array(buffer);
  socket.send(intview);

  var blob = new Blob([buffer]);
  socket.send(blob);
}
```

- WebSockets übertragen entweder binär oder UTF-8 kodiert – andere Format müssen in der Anwendung vereinbart werden

Versenden und Verarbeitung von Daten

- Da es sich um eine asynchrone Schnittstelle handelt kehrt ein Aufruf unmittelbar zurück – egal ob Daten schon versandt sind oder nicht
- Um zu ermitteln wie viele Daten noch auf Browser-Seite auf Versand warten kann man `bufferedAmount` abfragen
- Mit dieser Information kann man auch Priorisierung implementieren

```
1  var ws = new WebSocket('wss://example.com/socket');
2
3  ws.onopen = function () {
4    subscribeToApplicationUpdates(function(evt) {
5      if (ws.bufferedAmount == 0)
6        ws.send(evt.data);
7    });
8  };
```

Vereinbaren eines Protokolls

- Wie sollen sich Browser und Server über das ausgetauschte Nachrichtenformat einigen?
- Implizites Verständnis (bei Seiten kennen das Format bei Beginn der Kommunikation)
- Aushandeln eines Protokolls – wird von WebSockets unterstützt

```
1  var ws = new WebSocket('wss://example.com/socket',
2                         ['appProtocol', 'appProtocol-v2']);
3
4  ws.onopen = function () {
5    if (ws.protocol == 'appProtocol-v2') {
6      ...
7    } else {
8      ...
9    }
10 }
```

Etablierung einer Websocket-Verbindung

- WebSockets eröffnen eine Reihe von Möglichkeiten und Funktionen und entsprechend muss bei Verbindungsaufbau ein entsprechender Informationsaustausch stattfinden
- HTTP und die etablierte Infrastruktur wird dazu genutzt
- Konkret wird HTTP Upgrade um einige spezifische Header ergänzt
  - `Sec-WebSocket-Version` – Version welche durch den Client unterstützt wird. Aktuell "13"(siehe RFC6455)
  - `Sec-WebSocket-Key` – Automatisch erzeugter Schlüssel als Mechanismus zum Verbindungsaufbau (nicht sicherheitsrelevant!)
  - `Sec-WebSocket-Accept` — Antwort des Servers bzgl. des Schlüssels
  - `Sec-WebSocket-Protocol` – Festlegung eines Subprotokolls, Client sendet Liste, Server antwortet mit ausgewähltem Protokoll
  - `Sec-WebSocket-Extensions` – Mechanismus zum aushandeln von Erweiterungen

Etablierung einer Websocket-Verbindung

- Anfrage

```
GET /socket HTTP/1.1
Host: thirdparty.com
Origin: http://example.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Protocol: appProtocol, appProtocol-v2
Sec-WebSocket-Extensions: x-webkit-deflate-message, x-custom-extension
```

- Antwort

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Access-Control-Allow-Origin: http://example.com
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: appProtocol-v2
Sec-WebSocket-Extensions: x-custom-extension
```

- Cross-Origin Resource Sharing (CORS) wird ebenfalls behandelt

Zusammenfassung und Vergleich

- Jeder der Technologien hat sein unmittelbares Anwendungsgebiet
- WebSockets bietet eine bidirektional stromorientierte Kommunikation – vergibt aber Mechanismen des Browsers

	XMLHTTPRequest	**Server-Sent Events**	**WebSocket**
Request streaming	no	no	yes
Response streaming	limited	yes	yes
Framing mechanism	HTTP	event stream	binary framing
Binary data transfers	yes	no (base64)	yes
Compression	yes	yes	limited
Protocol	HTTP	HTTP	WebSocket
Message Overhead	HTTP/1.x 500-800 byte HTTP/2 min. 8 byte	5 byte	2-14 byte per frame

Referenzen

- Vorlesung ist an Kapitel 15, 16 und 17 des Buchs angelehnt

Literatur

[1] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, Inc., 2013.