3 Höhere Interaktivität und Echtzeitfähigkeit

Durch das besprochene Request-/Response-Modell, in dem immer der Client die Anfrage initiiert und eine darauf zurückgelieferte Antwort vom Server zur Folge hat, dass die dargestellte Webseite vollständig neu gerendert wird, ist die Umsetzung von Webanwendungen mit einer höheren Interaktivität und Echtzeitfähigkeit zunächst nicht realisierbar. Hierfür bedarf es zusätzlicher Mechanismen, mit denen diese Anforderungen bedient werden können.

Fragen, die dieses Kapitel beantwortet:

- Wie sind die Anforderungen an eine höhere Interaktivität und Echtzeitfähigkeit von Webanwendungen über die Zeit in neue Ansätze und Technologien gemündet?
- Was für Verfahren sind daraus hervorgegangen (im Wesentlichen XMLHttpRequest, Polling, Long-Polling und COMET/HTTP-Streaming) und wie funktionieren diese?
- Wie haben sich schließlich die aktuellen Ansätze in der HTML5-Standardfamilie Server-Sent Events und WebSockets daraus entwickelt?

3.1 XMLHttpRequest (XHR)

Durch den Einzug von XMLHttpRequest (XHR) [KASS14] in den Browser steht dem Webentwickler eine vom W3C standardisierte API zur Verfügung, mit der JavaScript-gesteuert HTTP-Requests abgesetzt werden können. Der JavaScript-Codeschnipsel in Listing 3.1 zeigt Ihnen exemplarisch, wie dies für eine GET-Anfrage erfolgt.

Listing 3.1 Exemplarische GET-Anfrage eines JavaScript-gesteuerten HTTP-Requests

```
1 var xhr = new XMLHttpRequest();
2 xhr.open("GET", "http://www.example.com", true);
3 xhr.onreadystatechange = function() {
4     if(this.readyState == 4 && this.status == 200) {
5         console.log(this.responseText);
6     }
7 }
8 xhr.send();
```

Als Erstes brauchen Sie ein `xhr`-Objekt, das Sie durch den Konstruktoraufruf – wie in Zeile 1 abgedruckt – erzeugen können. Steht Ihnen eine Instanz eines `xhr`-Objekts zur Verfügung, können Sie darüber HTTP-Anfragen absetzen. In Zeile 2 wird eine `GET`-Anfrage an den Host `www.example.com` vorbereitet. Damit die Kommunikation den Programmablauf nicht blockiert, registrieren Sie als Nächstes eine Funktion, die aufgerufen werden soll, sobald sich der Zustand, in dem sich das `xhr`-Objekt befindet, ändert. Hat sich dieser geändert, können Sie einige Dinge prüfen. Im Beispiel wird geprüft, ob sich das Objekt im Zustand mit der Nummer `4` befindet. Wenn Sie im genannten W3C-Standard nachschauen, sehen Sie, dass dieser Zustand angenommen wird, sobald die ganze Antwort des Servers auf der Clientseite eingegangen ist und verarbeitet wurde. Ist dem so und war die `GET`-Anfrage zudem erfolgreich (der Statuscode in der Response ist `200`), werden die empfangenen Nutzdaten in der JavaScript-Konsole ausgegeben. Bis jetzt ist aber noch nichts passiert. Die Kommunikation wurde bisher nur vorbereitet. Der entsprechende Request wird erst durch das Aufrufen der `send()`-Methode ausgelöst. Die Kommunikation läuft dann asynchron ab, was bedeutet, dass der Programmablauf durch die Kommunikation nicht blockiert wird. Andere JavaScript-Komponenten auf der aktuell angezeigten Webseite werden also nicht beeinflusst und laufen wie gewohnt weiter. Diese Möglichkeit, asynchrone Anfragen an den Server stellen zu können, ohne dass die dargestellte Webseite blockiert oder gar neu geladen bzw. neu gerendert wird, bildet die technische Grundlage für moderne Web-Frontends, die auch unter dem Oberbegriff Ajax (*Asynchronous JavaScript and XML*) geführt werden.

Bild 3.1 veranschaulicht den Lebenszyklus eines XHR-Objekts und die Zustände, die es dabei durchlaufen kann. Einen Überblick über die Konstruktoren, Methoden, Attribute und Events eines XHR-Objekts finden Sie in Anhang A.

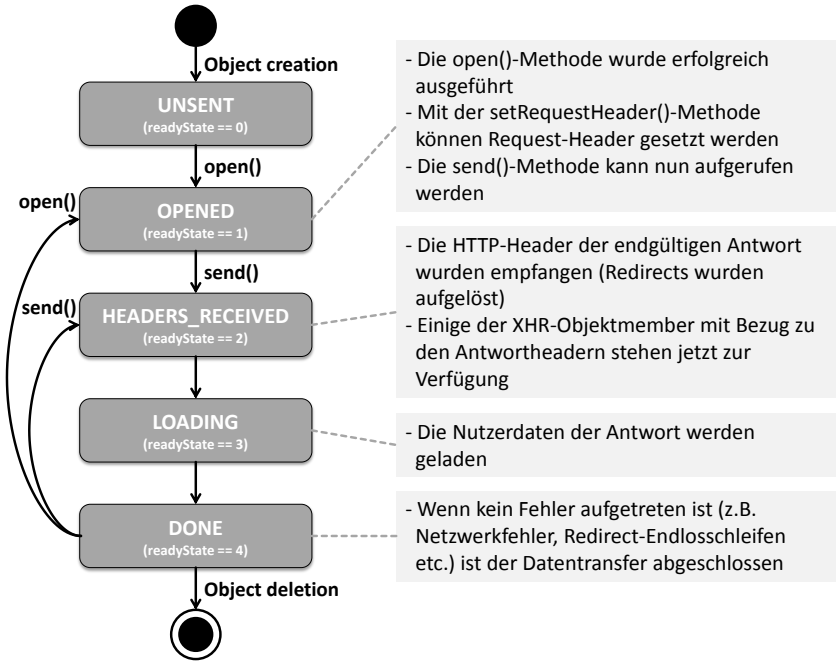

Bild 3.1 UML-Zustandsdiagramm eines XHR-Objekts gemäß dem W3C Working Draft [KASS14]

3.2 Polling

Viele Echtzeit-Webanwendungen werden heute durch *Polling* oder Long-Polling realisiert. Beim einfachen Polling werden in einem festgelegten Zeitabstand (z. B. alle zwei Sekunden) Anfragen an den Server gestellt, auf die umgehend geantwortet wird. Das zugrunde liegende Prinzip können Sie an dem folgenden JavaScript-Codefragment nachvollziehen:

Listing 3.2 Grundprinzip von Polling

```
1 function poll() {
2     xhr.send();
3     clearTimeout(timeoutId);
4     timeoutId = setTimeout(poll(), 2000);
5 }
6 timeoutId = setTimeout(poll(), 2000);
```

Das Objekt `xhr` ist so instanziiert und initialisiert worden wie zuvor beschrieben. Der Unterschied besteht hier im wiederholten Aufrufen der `send()`-Methode, durch die `setTimeout()`-Funktion. Auf diese Weise können Sie eine Ressource auf dem Server periodisch anfragen und die Antwort auf der Clientseite verarbeiten, was ggf. eine Aktualisierung der dem Benutzer angezeigten Daten mit sich bringt.

Haben sich seit der letzten Anfrage keine Änderungen mehr ergeben, so wird dies durch eine leere Antwort signalisiert. Bild 3.2 zeigt ein Polling-Kommunikationsszenario, aus dem Sie die Funktionsweise und Eigenschaften des Ansatzes ablesen können.

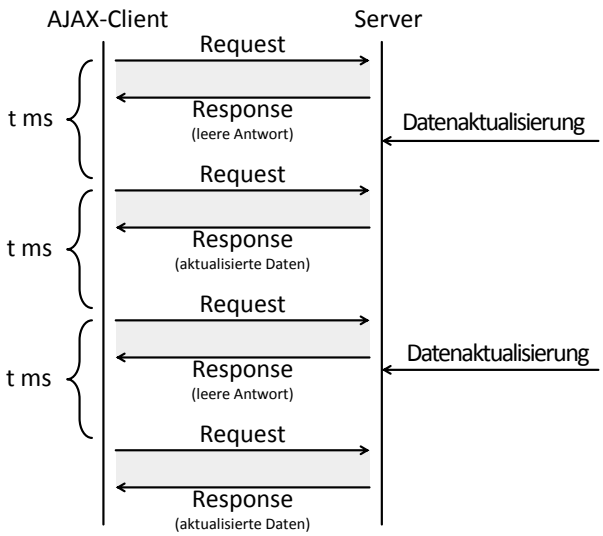

Bild 3.2 Polling

Polling ist einfach und browserdeckend zu implementieren. Der sich damit einstellende Overhead, der durch unnötige Anfragen und den damit verbundenen Kommunikations-

und Verarbeitungsaufwand zu Buche schlägt, ist allerdings nicht zu unterschätzen. Bei einer Entscheidung für oder wider Polling gilt es insbesondere, die benötigte zeitliche Auflösung zu berücksichtigen. Für Internetauktionen und Chat-Systeme muss diese naturgemäß recht fein sein, damit sich Veränderungen am Gebot rechtzeitig oder abgesetzte Nachrichten zeitnah auf dem Bildschirm der Clients einfinden. Derartige Anforderungen an die Aktualität der im Browser dargestellten Daten sind mit Polling nur durch einen erhöhten Ressourceneinsatz zu bedienen. Ein Dashboard zur Kontrolle des Wasserbedarfs von Grünpflanzen hingegen, kann ohne Weiteres effizient mit Polling implementiert werden. Hier können die Zeitintervalle großzügig definiert werden, da es keine gravierenden Auswirkungen hat, wenn z. B. im Stundenrhythmus ein Polling-Request abgesetzt wird und die darin evtl. enthaltenen Daten im ungünstigsten Fall fast eine Stunde alt sind. Das sollte weder den Server noch die Grünpflanzen zu sehr beanspruchen.

3.3 Long-Polling

Um den Overhead des Polling-Verfahrens zu reduzieren, tritt das *Long-Polling* mit der Abwandlung an, dass der Server nur dann prompt antwortet, wenn auch eine Änderung vorliegt, die er an den Client kommunizieren kann. Liegt nichts Neues vor, so antwortet er zunächst nicht und wartet eine festgelegte Zeit bei offener Verbindung auf zwischenzeitlich eintreffende Neuigkeiten (siehe Bild 3.3).

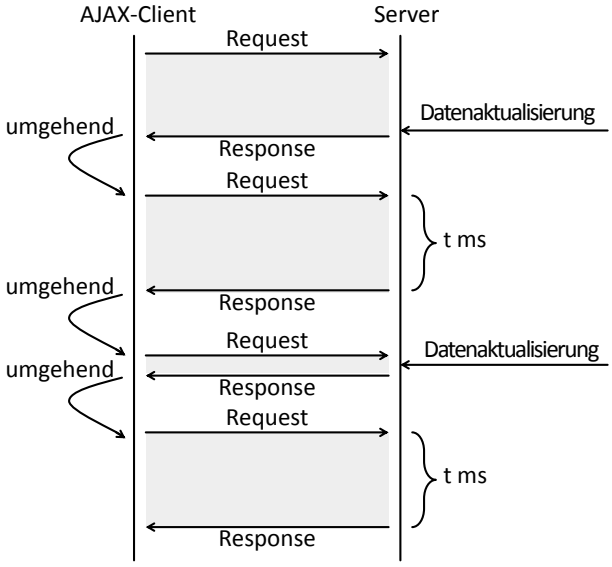

Bild 3.3 Long-Polling

Sobald Änderungen im Wartezustand eintrudeln, werden diese direkt über die offene Verbindung in der Response mitgeteilt und die Verbindung wie gewohnt geschlossen. Kommt

es zu keinen Änderungen bzw. Neuigkeiten im Wartezustand, so wird nach Ablauf der vordefinierten Wartezeit eine leere Antwort geschickt und die Verbindung geschlossen. Der Client öffnet in beiden Fällen postwendend eine neue Long-Polling-Verbindung, sodass der Server über eine quasi andauernde Verbindung zum Client verfügt.

3.4 Comet

Mit Comet [Rus06], dem zweiten gebräuchlichen Haushaltsputzmittel in den USA neben Ajax, steht ein Programmiermodell zur Verfügung, das ebenfalls die Long-Polling-Mechanismen beinhaltet, um Serverbenachrichtigungen in Webanwendungen zu integrieren, und ergänzt diese durch einige zusätzliche Spielarten. Comet sieht z. B. die Methode vor, anstelle von XMLHttpRequest JSONP [Sim10] zu verwenden. Die Idee hierbei ist, ein `<script>`-Element dynamisch zu erzeugen und dem DOM der geladenen Webseite hinzuzufügen. Das angefragte Skript enthält dabei vornehmlich JSON-Daten, die geeignet verpackt werden müssen, um sie mit JavaScript verarbeiten zu können. Die Verarbeitung wird durch Events initiiert, die im geladenen Skript ebenfalls enthalten sind. Sind die Daten lokal im Client verarbeitet, so wird zu guter Letzt, dem Polling-Muster entsprechend, das nächste Script-Element generiert und der identische Prozess erneut durchlaufen. Interessant an dieser als *Script Tag Long Polling* bekannten Methode ist, dass die JSON-Daten über das `src`-Attribute des `<script>`-Elements von einer (Sub-)Domain „gepollt“ werden können, die sich von der geladenen Webseite unterscheiden kann. Damit zählt das Script Tag Long Polling zu den wenigen Verfahren, die die Same Origin Policy des Webbrowsers zu überwinden vermögen, was durch CORS (Cross-Origin Resource Sharing) [Kes14] abgelöst werden soll. Auch in der neuen Version des XHR-Objekts im Browser – XHR in Level 2 – sind Mechanismen definiert, mit denen die Prüfung des Ursprungsortes einer Ressource selektiv außer Kraft gesetzt werden kann.

Außerdem sieht Comet auch Varianten vor, die auf persistenten HTTP-Verbindungen (engl. *Persistent Connection*) aufbauen. Der Webserver schließt hierbei die TCP-Verbindung nicht umgehend nach Beantwortung eines HTTP-Requests, sondern hält diese eine vorbestimmte Zeit offen, damit der TCP-Kanal zur Abarbeitung weiterer HTTP-Anfragen des Browsers an den Server verwendet werden kann. Dies ist insbesondere für Hypertext-Dokumente wie HTML sinnvoll, die eine Vielzahl von Ressourcen vom Server laden müssen, damit sie vollständig gerendert und angezeigt werden können. Nachrichtenseiten wie z. B. heise[1] können dabei leicht auf über 100 Ressourcen pro Seite kommen (Bilder, Skripte, Styles, Schriften usw.). Gesteuert wird dies über den HTTP-Header `Connection`. Trägt dieser den Wert `close`, so wird der TCP-Kanal nach Auslieferung der Response geschlossen (Default für HTTP 1.0). Ist der Wert des Connection-Headers auf `keep-alive` gesetzt, bleibt die TCP-Verbindung geöffnet (Default für HTTP 1.1). Eine aufrechterhaltene Verbindung kann von einer Comet-Umgebung verwendet werden, um mit einem unsichtbaren Inline-Frame, der kontinuierlich mit JavaScript-Fragmenten gefüllt wird, einen Server-Push zu realisieren. Die tröpfchenweise eintreffenden JavaScript-Anweisungen werden

[1] *www.heise.de/*

vom Interpreter im Browser sukzessive ausgeführt und können somit zur servergesteuerten Aktualisierung des Clients herangezogen werden.

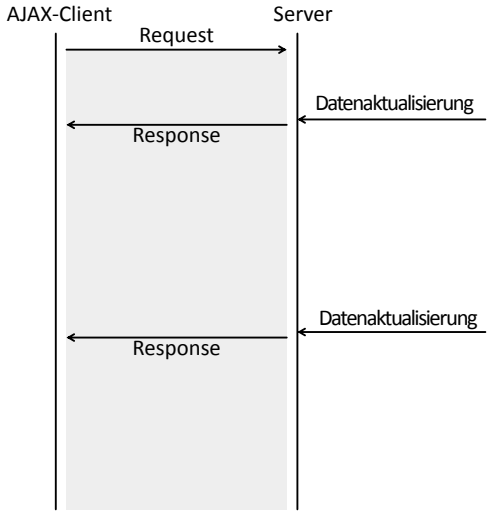

Bild 3.4 HTTP-Streaming

Einen großen Nachteil von Comet können Sie in der Tatsache finden, dass es sich hierbei nicht um einen Standard handelt. Das Fehlen von verbindlichen Spezifikationen steht häufig einer breiten Verwendung entgegen. Um dem zu begegnen, ist mit der Definition des Bayeux-Protokolls [RWDN07] eine Initiative ins Leben gerufen worden, die eine entsprechende Norm für die Entwicklung Comet-basierter Webbenachrichtigungen festlegt, die Entwicklern in erster Linie eine leichtere Umsetzung und höhere Interoperabilität gewährleisten soll. Um insbesondere den letzten Punkt realisieren zu können, hat die Dojo Foundation im Rahmen des CometD-Projekts [The14] eine Reihe von Implementierungen des Bayeux-Protokolls in verschiedenen Programmiersprachen – darunter z. B. JavaScript, Java, Perl und Python – bereitgestellt. Da sich das Bayeux-Protokoll allerdings nicht den Einzug in etablierte Standardisierungsorganisationen bahnen konnte, es aber nunmehr alternative Entwicklungen in der IETF und dem W3C gibt, empfehlen wir Ihnen, sich auf Letztere zu konzentrieren. Das W3C fokussiert sich dabei auf Standardisierungsarbeiten rund um JavaScript-APIs für die HTML5-Standardfamilie. In unserem Kontext sind die Kommunikationserweiterungen von Interesse, die in der Gruppe HTML5 Connectivity zusammengefasst sind. In dieser Kategorie finden Sie die Standards Server-Sent Events (SEE, siehe Abschnitt 3.5) und WebSockets (siehe Bild 1.2 und Bild 3.5).

Für jede Gruppe von inhaltlich verwandten Standards gibt es neben einem Gruppennamen auch ein eigenes Logo. Diese Logos für die verschiedenen als *Technology Classes* bezeichneten Gruppen können Sie Bild 3.6 entnehmen. Weitere Informationen hierzu und über die Möglichkeit, Ihr eigenes HTML5-Logo zusammenzustellen, haben Sie auf der extra dafür veröffentlichten Webseite des W3C.[2]

[2] *http://www.w3.org/html/logo/*

Bild 3.5 HTML5 Connectivity-Logo [W3C14]

Bild 3.6 HTML5-Logo mit allen Technology Classes [W3C14]

3.5 Server-Sent Events

Die bewährten Teile aus den aufgezeigten Server-Push-Technologien sind in HTML5 im Server-Sent Event (SSE) [Hic12a] Standard verschmolzen worden. SSE definiert eine API, mit der sich HTTP-Verbindungen herstellen lassen, die der Webserver zum spontanen Senden an den Client verwenden kann. In der W3C „Candidate Recommendation“ ist dafür das `EventSource`-Objekt eingeführt und seine Schnittstelle spezifiziert worden. Ein `EventSource`-Objekt repräsentiert die Clientseite, die Benachrichtigungen empfangen kann. Die Benachrichtigungen werden in Form von DOM-Events an die Anwendung weitergegeben. Es werden drei Event-Handler spezifiziert, die vom Browser entsprechend anzubieten sind. Der `onopen`-Handler wird aufgerufen, wenn die SSE-Verbindung geöffnet worden ist. Tritt ein Fehler auf, wird der `onerror`-Handler aktiviert. Das Eintreffen von Servernachrichten wird durch den `onmessage`-Event signalisiert und an die registrierte Funktion zur Weiterverarbeitung weitergereicht (siehe Listing 3.3).

Listing 3.3 Aufbau eines SSE-Kanals und das Lauschen auf Serverbenachrichtigungen

```
1 <html>
2     <head>
3         <script>
4             var stats = new EventSource('stats');
5             /*
```

```
            * Für jede eintreffende Nachricht vom Server
            * wird die hier registrierte Funktion aufgerufen
            * und ausgeführt.
            */
            stats.onmessage = function (event) {
                statsDiv = document.getElementById('stats');
                statsDiv.innerHTML += "<br>" + event.data;
            };
        </script>
    </head>
    <body>
        <div id="stats"></div>
    </body>
</html>
```

Wird die in Listing 3.3 dargestellte Webseite in einen Browser geladen, so wird durch das Erzeugen des `EventSource`-Objekts ein HTTP GET-Request an die im Aufruf angegebene URL abgesetzt, der den SSE-Kanal etabliert (siehe Listing 3.4). Für den SSE-Request sind keine spezifischen HTTP-Header zwingend erforderlich. Der Working Draft rät allerdings, den für SSE registrierten MIME-Typ `text/event-stream` im `Accept`-Header einzutragen, das Caching der SSE-Requests zu unterbinden sowie sicherzustellen, dass es sich um eine persistente Verbindung handelt (`Connection: keep-alive`).

Listing 3.4 Wesentliche Komponenten des GET-Requests, der bei der Erzeugung eines EventSource-Objekts generiert wird

```
GET /stats HTTP/1.1
[...]
Accept: text/event-stream
Last-Event-Id: 6
Cache-Control: no-cache
Connection: keep-alive
```

Auf der Serverseite muss nicht viel getan werden. Wesentlich ist hier, dass die Verbindung zum Client nicht geschlossen wird und dass die definierten Felder in der vorgegebenen Syntax in die offene Response geschrieben werden. Listing 3.5 zeigt eine dortige (noch offene) Response. Auch in diesem Beispiel sind nur die für SSE relevanten Header enthalten. Der `Content-Type` muss auf den für SSE registrierten MIME-Type `text/event-stream` gesetzt sein und das Caching der SSE-Response unterbunden werden.

Listing 3.5 Offene HTTP-Response, die es dem Server erlaubt, spontane Nachrichten an den Client zu versenden

```
HTTP/1.1 200 OK
[...]
Content-Type: text/event-stream
Expires: Mon, 01 Jan 2015 00:00:00 GMT
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Connection: close
```

```
id: 7
data: {"servername": "tatooine", "cpu-load": "45", "time": "14:33:48"}

id: 8
data: {"servername": "tatooine", "mem-usage": "78", "time": "14:33:48"}

... weitere Serverbenachrichtigungen können folgen ...
```

In der Response aus Listing 3.5 sind zwei Events enthalten, die in dem Beispiel Werte zur Auslastung von Servern bereitstellen. Gemäß den Definitionen zum `text/event-stream` MIME-Type besteht ein Event aus keiner, einer oder mehreren Kommentarzeilen, die mit einem Doppelpunkt beginnen und einer oder mehreren Zeilen mit Datenfeldern. Datenfelder sind durch einen Doppelpunkt getrennte Name-Wert-Paare, wobei aktuell die vier Namen `event`, `data`, `id` und `retry` spezifiziert sind. Andere Feldnamen werden ignoriert. Als Wert kann ein beliebiger Textstring eingesetzt werden. In Listing 3.5 sehen Sie z. B. einen JSON-formatierten String, der die Messdaten bereitstellt. Events werden durch eine Leerzeile voneinander getrennt.

Durch die Zugehörigkeit zur HTML5-Standardfamilie ist eine breite Browserunterstützung für SSE zu erwarten. Tatsächlich hat gemäß „Can I use“ auch ein Großteil der neueren Browser SSE mit an Bord [Dev14a]. Die einzige Ausnahme stellt hier der Internet Explorer dar, der SSE bisher nicht unterstützt. Laut dem angegebenen Status auf der Internetseite modernIE stehen die SSE für die kommende Version 12 des Microsoft User Agents zur Diskussion.[3] Als Fallback kann auf die eingangs diskutierten wegbereitenden Polling-Spielarten zurückgegriffen werden, was insbesondere für die Internet Explorer-Unterstützung vorgesehen werden muss. Abschließend kann festgehalten werden, dass mit SSE ein unidirektionaler Kanal vom Server zum Client aufgebaut werden kann, der für Anwendungsfälle interessant ist, die eine Serverbenachrichtigungsfunktion benötigen. Das Erklingen eines freundlichen „Sie haben Post“-Hinweises ist ein Paradebeispiel für den Einsatz von SSE. Weitere sind Newsfeeds, Finanzdaten- und Sportticker sowie Monitoring von Haushalt und Industrie.

3.6 Bewertung der Verfahren

Polling bzw. Long-Polling hat unter anderem den Nachteil, dass pro Request-/Response-Nachrichtenpaar jeweils ein Header benötigt wird, was zu sehr hohen Datenmengen führen kann. Hat der Server keine neuen Daten, wird eine informationslose Nachricht mit einem Request- und Reponse-Header erzeugt.

Trotz der angerissenen Variantenvielfalt bleiben Nachteile dieser konstruierten Lösungen für Serverbenachrichtigungen bestehen. Im Detail finden sich spezifische Schwächen hinsichtlich der Kontrolle der Kommunikationsverbindung, der Behandlung von Fehlern und der Sicherheit. Zudem bereitet der bunte Strauß an herangewachsenen Technologien nicht

[3] *https://status.modern.ie/serversenteventseventsource?term=Server-Sent%20Events*

gerade einen geradlinigen Einstieg in die Thematik, da es zunächst darum geht, ein Verständnis für den Kessel Buntes zu entwickeln. Der Hauptnachteil ist und bleibt aber im kontinuierlichen Anfragen von Daten begründet. Ein besserer Ansatz würde eine tatsächliche Verbindung etablieren, die vom Server genutzt werden kann, anstatt eine scheinbare Verbindung durch das sich stetig wiederholende Anfragen des Servers zu emulieren. Eine derartige Technologie ist durch WebSockets gegeben, die in der HTML5-Standardfamilie enthalten sind und denen wir für die restliche gemeinsame Zeit mit diesem Buch unsere volle Aufmerksamkeit zukommen lassen.