5 Der Client: Die W3C WebSocket-API

Damit Sie eine WebSocket-Verbindung zwischen Client und Server aufbauen und damit das Protokoll, das wir uns im vorherigen Kapitel zu Gemüte geführt haben, anstoßen können, stellt Ihnen ein entsprechender W3C-Standard die dafür notwendige JavaScript-API [Hic12b] bereit. Hieran lässt sich schön die inhaltliche Trennung der Standardisierungsgremien IETF und W3C sehen. Die IETF standardisiert Internet-Protokolle, wozu die Protokollspezifikation der WebSockets gehört (siehe Kapitel 4). Das W3C hingegen spezifiziert Webtechnologien, die insbesondere den Browser angehen. Hierzu zählt folglich die API-Spezifikation des WebSocket-Standards, die im Browser implementiert sein muss und die wir in diesem Kapitel genauer betrachten möchten.

Fragen, die dieses Kapitel beantwortet:

- Wie wird ein WebSocket-Objekt instanziiert?
- In welchen Zuständen kann es sich während seines Lebens befinden?
- Wie kann man darüber Daten senden und empfangen?
- Wie geht man mit Fehlern um?

5.1 Was bisher geschah

Alle Entwürfe vom ersten bis zum letzten Working Draft sowie die Candidate Recommendation und die Editor's Drafts wurden verfasst vom Google-Mitarbeiter Ian Hickson. Ian Hickson verfasste auch den ersten Internet-Draft des WebSocket-Protokolls am 9. Januar 2009.

Der erste Working Draft der WebSocket-API erschien am 23. April 2009. Am 29. Oktober 2009 wurde bereits der zweite Working Draft herausgegeben. In dieser Version wurde der Konstruktor durch einen zweiten optionalen Parameter ergänzt, worin ein Subprotokoll übergeben werden konnte. Des Weiteren wurde die API überarbeitet. Für das Versenden von Nachrichten und das Schließen der Verbindung wurden in der ersten Version noch die Methoden `postMessage()` und `disconnect()` verwendet. Diese Methoden wurden im zweiten Working Draft zu `send()` und `close()`. Es wurde außerdem ein Attribut namens

`bufferedAmount` für das WebSocket-Objekt hinzugefügt. Damit kann die Anzahl an Bytes ausgelesen werden, die noch nicht versendet wurden.

Danach folgten zwei weitere Working Drafts am 22. Dezember 2009 und am 11. April 2011. Die Version des 11. April 2011 wurde durch den Event-Handler `onerror` ergänzt. Zusätzlich kann seitdem ein zweiter optionaler Parameter genutzt werden, wodurch mehrere Subprotokolle in Form eines String-Arrays angegeben werden können statt bis dato nur eins. Somit kann ein Server eines der Subprotokolle auswählen, das er unterstützen will. Aus diesem Grund wurde zusätzlich das Attribut `protocol` eingeführt. Der Client ist dadurch in der Lage zu erkennen, welches dieser Subprotokolle vom Server unterstützt wird. Die Anzahl der Ready States wurde im April 2011 durch `CLOSING` ebenfalls um eins erhöht. Vorher existierten nur die drei Verbindungszustände `CONNECTING`, `OPEN` und `CLOSED`. In dieser Version konnte auch zum ersten Mal bei einer Beendigung der Verbindung ausgelesen werden, ob die Verbindung sauber geschlossen wurde.

Am 29. September desselben Jahres erschien der nächste Working Draft. Dieser Arbeitsentwurf ermöglichte es, Binärdaten in Form von BLOBs und ArrayBuffern zu versenden. Die bisherigen Versionen hatten nur Strings vorgesehen. Da jetzt zwei Arten von Binärdaten versendet und empfangen werden konnten, hat das W3C in dieser Version das Attribut `binaryType` eingeführt. Der Client kann dadurch festlegen, in welcher Form er die Binärdaten empfangen will. Bei der `close()`-Methode kam die Möglichkeit hinzu, optional einen Close-Code als Zahl und einen Grund für die Beendigung als String an den Server mitzugeben. Zusätzlich wurde noch das Attribut `extensions` hinzugefügt.

Im Dezember 2011 wurde die erste Candidate Recommendation der WebSocket-API veröffentlicht. Danach folgten zwei weitere Working Drafts am 24. Mai 2012 und am 9. August 2012. Die zweite Candidate Recommendation wurde am 20. September 2012 vorgestellt. Darauf folgte am 04. Juni 2014 der letzte Editor's Draft.

5.2 Browserunterstützung

WebSockets sind fester Bestandteil der HTML5-Standardfamilie. Als solcher ist der Support in den aktuellen Versionen der fünf Freunde schon recht breit. Die Website

http://www.caniuse.com/

(gesprochen: can i use <dot> com) gibt über die Browserunterstützung vieler neuer Webtechnologien – insbesondere im HTML5-Umfeld – Auskunft. Tabelle 5.1, die einen Teil der Kompatibilitätstabelle von caniuse.com zur WebSockets-Unterstützung zeigt, macht einem Mut. Sie zeigt, dass alle relevanten Browser sowohl im Desktop- als auch im Mobile-Bereich WebSockets unterstützen. Die grauen Versionsnummern bedeuten eine teilweise Unterstützung, wohingegen die weiß markierten Browserversionen nicht mit WebSockets umgehen können.

Von Hand können Sie auch schnell herausfinden, ob ein Browser WebSockets unterstützt. Hat er das Protokoll und die API implementiert, dann finden Sie im `window`-Objekt ein `WebSocket`-Element. Dieses fehlt entsprechend, wenn der Browser keine native Unterstüt-

Tabelle 5.1 Browserunterstützung für WebSockets [Dev14c]

IE	Firefox	Chrome	Safari	Opera	iOS Safari	Android-Browser	Opera Mobile	Blackberry-Browser
		4						
		5						
	2	6						
	3	7						
	3.5	8						
	3.6	9						
	4	10						
	5	11						
	6	12						
	7	13						
	8	14						
	9	15						
	10	16						
	11	17						
	12	18						
	13	19		9.6				
	14	20		10.1				
	15	21		10.5				
	16	22		10.6				
	17	23		11				
	18	24		11.1				
	19	25		11.5				
	20	26		11.6				
	21	27		12				
	22	28		12.1				
	23	29		15				
	24	30	3.1	16				
	25	31	3.2	17		2.1		
	26	32	4	18	3.2	2.2		
5.5	27	33	5	29	4.1	2.3		
6	28	34	5.1	20	4.3	3		
7	29	35	6	21	5.1	4	10	
8	30	36	6.1	22	6.1	4.1	11.5	
9	31	37	7	23	7.1	4.3	12	
10	32	38	7.1	24	8	4.4	12.1	7
11	33	39	8	25	8.1	4.4.4	24	10
	34	40		26		37		
	35	41		27				
	36	42						

zung für WebSockets mit an Bord hat. Somit ist der Test mit den JavaScript-Codezeilen aus Listing 5.1 getan (unbedingt auf die Groß- und Kleinschreibung achten!).

Listing 5.1 Test auf WebSocket-Unterstützung in einem Browser

```
<html>
    <body>
        <script type="text/javascript">
            if ("WebSocket" in window) {
                alert("Super! WebSockets werden unterstuetzt!");
            }
            else{
                alert("Schade! Wir muessen uns um eine Alternative
                    kuemmern.");
            }
        </script>
    </body>
</html>
```

In den meisten Fällen sollte ein Check auf WebSocket-Support im Browser ausreichen, um loslegen zu können. Es können sich bei erfolgreichem Check dennoch Problemchen einschleichen. Der Teufel steckt im Detail. Dies kann Ihnen passieren, wenn ältere Protokollimplementierungen auf der Client- oder Serverseite zum Einsatz kommen. Welche Version des Protokolls Sie wo antreffen können, haben wir in Tabelle 5.2 zusammengetragen.

Tabelle 5.2 Browserunterstützung in Bezug auf WebSocket-Protokollversionen [sta14, Aut12]

WebSocket Protokoll-version	Chrome	Firefox	IE	Opera	Opera Mobile	Safari
Hixie-75	4.0, 5.0	–	–	–	–	5.0.0
HyBi-00/ Hixie-76	6.0 - 13.0	4.0 (standard-mäßig deaktiviert)	–	11 (standard-mäßig deaktiviert)	–	5.0.2, 5.1
HyBi-07+	14.0	6.0 (Präfix: MozWeb-Socket)	9.0 (Silverlight Erweiterung)	–	–	–
HyBi-10	14.0, 15.0	7.0 - 10.0 (Präfix: MozWeb-Socket)	10 (Windows 8 Developer Preview)	–	–	–
HyBi-17/ RFC 6455	16	11	10	12.10	12.10	7

Sie sollten darauf achten, dass die Komponenten des Anwendungssystems, die Sie mit Ihren Implementierungen beeinflussen können, auf die letzte Version des Protokolls HyBi-17 bzw. RFC 6455 abstellt. Ihre Tests sollten unbedingt auch frühere Versionen der Browser mit

WebSocket-Support mit einschließen, damit Sie feststellen können, ob die Verwendung Ihrer Anwendung mit älteren Browsern funktioniert. Ergeben die Tests das Gegenteil, müssen für diese – ähnlich wie für Browser ohne WebSocket-Support – alternative Fallbacks vorgesehen und implementiert werden.

Wenn Ihnen die WebSocket-Browserabdeckung so nicht reicht und Sie höhere Anforderungen an die zu unterstützenden Browser haben, dann müssen Sie ebenfalls Fallbacks implementieren. Diese beruhen in der Regel auf den in Kapitel 3 genannten Verfahren wie Long-Polling oder Comet. Tatsächlich nehmen Ihnen viele clientseitige Frameworks für die Entwicklung mit WebSockets diese Arbeit bereits ab. Dazu an gegebener Stelle mehr (siehe Abschnitt 6.2.3).

5.3 Namensschema

Wie können Sie nun einen WebSocket-Server ansprechen? Wie Sie einen Webserver ansprechen, wissen Sie. Die URL spielt dabei eine wesentliche Rolle. In Kapitel 2 haben wir gesehen, wie eine URL allgemein aufgebaut ist und wie Sie eine solche für konkrete Ausprägungen von Webressourcen konstruieren müssen. Zur Adressierung eines WebSocket-Servers können wir auf diesem Wissen aufbauen und werden einige Déjà-vus erleben. In Kapitel 3 des RFC 6455 finden Sie dazu die folgende Spezifikation [FM11a]:

```
"ws:" "//" host [ ":" port ] path [ "?" query ]
```

Bei der Notation handelt es sich um die angereicherte Backus-Naur-Form (ABNF), die im RFC 3986 [CO08] definiert ist. Damit Sie die angegebene URL-Konstruktionsregel lesen und verstehen können, müssen Sie wissen, dass die eckigen Klammern optionale Terme bezeichnen, die in einer URL nicht zwingend auftauchen müssen. Die Terme in Anführungszeichen stellen Zeichenketten dar, die genau in der angegebenen Form in der URL eingefügt werden. Terme ohne Anführungszeichen sind variable Platzhalter, die mit entsprechendem Inhalt zu füllen sind. Sowohl der Platzhalter für den Host, die Portnummer und den Query-String kennen wir schon aus regulären Web-URLs. Eine einfache WebSocket-URL kann also z. B. wie folgt ausgestaltet sein:

ws://chat.example.com/

oder auch

ws://noch-deins-bald-meins.de/auktionen/

Einfach gesprochen, unterscheidet sich eine WebSocket-URL von einer Web-URL damit im Wesentlichen durch den angepassten Scheme, der nicht wie im Web üblich `"http:"`, sondern für WebSockets `"ws:"` ist.

Neben dieser URL-Spezifikation definiert der RFC 6455 noch eine zweite URL, die nach der folgenden ABNF aufgebaut ist:

```
"wss:" "//" host [ ":" port ] path [ "?" query ]
```

Wie Sie sehen, unterscheidet sich diese URL-Konstruktionsregel nur durch einen winzigen Zusatz im Vergleich zur vorherigen. Dieser kleine, aber feine Unterschied befindet sich im Scheme der URL und enthält ein zusätzliches „s“. Dieses steht für „secure“ und ist das Pendant zu `"https:"`. Dazu mehr, wenn wir in Abschnitt 7.2.2 sicherheitsrelevante Aspekte beleuchten.

5.4 Objekterzeugung und Verbindungsaufbau

Da die clientseitige Programmierung in der Regel in einem Webbrowser stattfindet, ist die hierfür definierte API eine vom W3C standardisierte JavaScript-API. Dreh- und Angelpunkt ist dabei eine Instanz eines WebSocket-Objekts. Eine solche können Sie mittels der bereitgestellten Konstruktoren erzeugen. Abstrakt geschrieben, stehen Ihnen die folgenden Konstruktoren zur Verfügung:

```
[Constructor(
    DOMString url, optional (DOMString or DOMString[]) protocols)]
```

Das ist die Schreibweise, die im Standard genutzt wird. Sie basiert auf der WebIDL [McC12], einer abstrakten Schnittstellenbeschreibungssprache, die Sie in vielen W3C-Standards antreffen. Eine kürzere Notation kann auf Grundlage der angereicherten Backus-Naur-Form (ABNF) [CO08] wie folgt angegeben werden:

```
WebSocket(url[, protocols])
```

Ob die eine oder die andere Schreibweise, aus beiden können Sie die drei verschiedenen Wege der Objekterzeugung ablesen, die Ihnen bereitstehen. Der minimale Konstruktor erwartet genau *ein* String-Argument, das ihm die URL des WebSocket-Endpunkts angibt. Möchten Sie z. B. eine Verbindung zum WebSocket-Server

ws://echo.websocket.org/

aufbauen, so müsste Ihr Konstruktoraufruf wie folgt aussehen:

```
var ws = new WebSocket('ws://echo.websocket.org/');
```

Sie können dem WebSocket-Konstruktor aber auch noch ein zweites String-Argument übergeben. Das erste bleibt wie gehabt die zwingend erforderliche URL des adressierten Servers. Das zweite kann optional das Subprotokoll angeben, das über den WebSocket-Kanal zwischen Client und Server verwendet werden soll. Für das folgende Beispiel wollen wir als Subprotokoll SOAP [GHM+07] verwenden:

```
WebSocket ws = new WebSocket('ws://echo.websocket.org/', 'soap');
```

Dies hat den Effekt, dass in die HTTP GET-Anfrage des WebSocket-Handshakes der folgende zusätzliche Headereintrag hinzugefügt wird:

```
Sec-WebSocket-Protocol: soap
```

Sie können unter Verwendung eines String-Arrays im zweiten Argument auch mehrere Subprotokolle angeben:

```
WebSocket ws =
    new WebSocket('ws://echo.websocket.org/', ['soap', 'wamp']);
```

Der Headereintrag der GET-Anfrage im WebSocket-Handshake hat dann entsprechend folgendes Aussehen:

```
Sec-WebSocket-Protocol: soap, wamp
```

Wie wir bereits in Abschnitt 4.2 besprochen haben, kommt es nur zu einer WebSocket-Verbindung, wenn der Server eines der angegebenen Subprotokolle akzeptiert. Welches Subprotokoll der Server schließlich gewählt hat, können Sie anhand eines Attributs, das wir in Abschnitt 5.8 behandeln, in Erfahrung bringen. Registrierte Subprotokolle sind in der WebSocket Subprotocol Name Registry der IANA nachzusehen [MML+14].

5.5 Zustände

Haben Sie auf diese Weise ein WebSocket-Objekt instanziiert, wurde der Handshake zwischen Client und Server durchgeführt. Je nachdem ob der Verbindungsaufbau erfolgreich war oder nicht, befindet sich Ihr Objekt in einem der im Standard dafür spezifizierten Zustände. Genauer gesagt sind es vier Zustände, die Sie in Tabelle 5.3 aufgelistet sehen.

Tabelle 5.3 Zustände und das `readyState`-Attribut gemäß [Hic12b]

Zustand	Beschreibung
CONNECTING (`readyState: 0`)	Das Objekt wurde instanziiert und befindet sich aktuell dabei, eine Verbindung zum Server aufzubauen.
OPEN (`readyState: 1`)	Die Verbindung ist erfolgreich aufgebaut und damit steht ein kommunikationsbereiter Kanal zur Verfügung.
CLOSING (`readyState: 2`)	Die Verbindung geht durch einen Closing-Handshake und wird damit geschlossen.
CLOSED (`readyState: 3`)	Die Verbindung ist geschlossen oder konnte gar nicht erst aufgebaut werden.

Zur besseren Veranschaulichung, was genau passiert und wie die Zustände zusammenhängen und ineinander übergehen, haben wir diese in Bild 5.1 in einem UML-Zustandsdiagramm zusammenhängend dargestellt.

Wie eingangs gesagt, befindet sich Ihr WebSocket, nachdem Sie ihn durch den Aufruf des Konstruktors erzeugt haben, zunächst für eine kurze Zeit im Zustand CONNECTING. Das `readyState`-Attribut Ihres Objekts hat dabei den Wert `0`. Im CONNECTING-Zustand versucht Ihr Objekt, eine Verbindung zum im Konstruktoraufruf angegebenen Server herzustellen. Die Objekterzeugung initiiert also den WebSocket-Handshake, wie wir ihn in Ab-

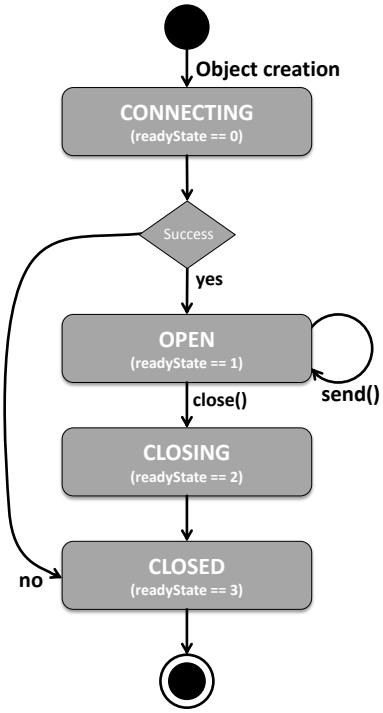

Bild 5.1 UML-Zustandsdiagramm eines WebSocket-Objekts gemäß [Hic12b]

schnitt 4.2 eingeführt und besprochen haben. Gelingt es Ihrem Objekt nicht, eine Verbindung zum Server herzustellen, so geht Ihr Objekt in den Zustand CLOSED über. Das `readyState`-Attribut bekommt in diesem Zustand den Wert `3` zugewiesen und ist im Grunde zu nichts mehr zu gebrauchen. Anderenfalls konnte die Verbindung hergestellt werden, was Ihr Objekt in den Zustand OPEN versetzt.

Im OPEN-Zustand ist Ihr Objekt sende- und empfangsbereit. Erkennen können Sie dies ebenfalls am Wert des `readyState`-Attributs. Dieses hat dann nämlich den Wert `1` inne. In diesem Zustand verweilt Ihr Objekt typischerweise eine längere Zeit, so lange wie Daten gesendet und/oder empfangen werden sollen. Sofern kein Fehler währenddessen auftritt, bleibt dieser Zustand so lange erhalten, bis einer der beiden Kommunikationspartner nicht mehr möchte und die Verbindung schließt. Auf der Clientseite erfolgt dies durch den Aufruf der einschlägigen Methode `close()`, für die es auf der Serverseite natürlich eine Entsprechung gibt. Kommt es dazu, dass eine bestehende WebSocket-Verbindung geschlossen wird, so geht Ihr Objekt zunächst in den Zustand CLOSING über.

Im CLOSING-Zustand verbleibt Ihr WebSocket-Objekt nur so lange, wie es dauert, den Closing-Handshake zu durchlaufen. Während dieser Zeit hat das `readyState`-Attribut den Wert `2`. Ist der Kanal geschlossen, geht Ihr Objekt in den vierten und letzten Zustand über.

Der CLOSED-Zustand markiert das Lebensende Ihres WebSocket-Objekts. An dieser Stelle können Sie nicht mehr viel damit tun. Wenn Sie eine weitere WebSocket-Verbindung aufbauen und verwenden wollen, müssen Sie dafür eine neue Instanz erzeugen. Dieses Objekt

können Sie also beruhigt durch das Setzen der Referenz auf `null` in die ewigen Jagdgründe verabschieden. Achten Sie beim Erreichen dieses Zustands unbedingt darauf, dass Ihre WebSocket-Objekte nicht mehr referenzierbar sind, damit der belegte Speicher frei gegeben werden kann und keine Speicherlöcher verbleiben.

5.6 Event-Handler

Die soeben besprochenen Zustände stehen im engen Zusammenhang mit den spezifizierten Event Handlern, mit denen auf Zustandsänderungen reagiert werden kann. Ändert sich der Zustand, in dem sich Ihr WebSocket-Objekt befindet, wird ein entsprechender Event ausgelöst, der wiederum den korrespondierenden Event Handler triggert. Überhaupt ist das Programmiermodell stark ereignisgetrieben. Tabelle 5.4 zeigt Ihnen die im Standard definierten Event Handler sowie die korrespondierenden Events.

Tabelle 5.4 Event Handler [Hic12b]

Event Handler	Event	Beschreibung
`onopen`	open	Der Event Handler `onopen` ist mit dem Zustand open assoziiert. Wird ein WebSocket-Objekt in den open-Zustand versetzt, wird der open-Event erzeugt, der von diesem Event Handler verarbeitet wird.
`onerror`	error	Der Event Handler `onerror` ist mit dem error-Event gekoppelt. Tritt ein Fehler beim Verbindungsaufbau, während der Datenübertragung oder beim Verbindungsabbau auf, wird dies über diesen Handler signalisiert und die Gründe für den Fehler werden in Ihre Anwendung getragen.
`onmessage`	message	Trifft eine Nachricht beim Client ein, wird der message-Event ausgelöst. Ihrer Anwendung wird dies mittels des Event Handlers `onmessage` signalisiert. Gleichzeitig werden die empfangenen Daten zur Weiterverarbeitung an Ihre Anwendung übergeben.
`onclose`	close	Der Event Handler `onclose` wird immer dann aktiviert, wenn die WebSocket-Verbindung geschlossen oder gar nicht erst zustande gekommen ist. Auch hier werden über den Event Handler Informationen an Ihre Anwendung übergeben, aus denen Sie die Gründe entnehmen können.

Die Event Handler sind Attribute des WebSocket-Objekts, denen Sie eine Funktion zuweisen können, die dann im Falle des Eintretens des zugehörigen Ereignisses ausgeführt wird. Sie müssen nicht für alle Event Handler eine Funktion angeben. Tatsächlich können Sie sich Anwendungen vorstellen, für die kein einziger der vier Event Handler eine Funktion zugewiesen bekommen muss. Dies trifft auf Anwendungen zu, die über einen WebSocket schlicht Daten vom Client zum Server schicken sollen wobei die umgekehrte Kommunikationsrichtung nicht benötigt wird. Beispiele für derartige Anwendungen lassen sich in den Settings finden, die den Client z. B. als eine Art Sensor nutzen und viele Daten mit

einer geringen Verzögerung kontinuierlich dem Server bereitstellen. Hierunter fallen z. B. Beobachtungen von Probanden im Rahmen von Usability-Tests, deren Interaktion mit einer Webseite über Mausbewegungsbeobachtungen sowie Video- und Audioaufzeichnungen erfasst werden. In der Regel sollte es aber zum guten Programmierstil gehören, für alle Eventualitäten entsprechende Codezeilen vorzusehen, die mit Fehlern umgehen und zum Schluss das Aufräumen übernehmen.

Wenn Ihr Programm direkt nach Etablierung der WebSocket-Verbindung etwas tun soll, dann können Sie das über den `onopen`-Event-Handler realisieren. Wenn Sie gerade keine Idee haben, was das sein könnte, möchten wir Ihnen mit dem folgenden Beispiel einige Anregungen geben.

Listing 5.2 Beispiel zur Verwendung des `onopen`-Event-Handler

```
1 ws.onopen = function() {
2     console.log('WebSocket-Verbindung aufgebaut.');
3     turnConnectionIndicatorOn();
4     ws.send('Add: ISIN=DE0008469008');
5 };
```

Sie könnten z. B. darüber Buch führen wollen, was alles mit Ihrem WebSocket-Objekt so passiert ist. Das ist sicherlich am einfachsten durch eine Logausgabe in der Konsole getan. Vielleicht haben Sie aber auch einen visuellen Indikator in die Webseite integriert, der dem Benutzer über den Zustand des Kommunikationskanals informiert. Diesen Indikator sollten Sie dann in der `onopen`-Funktion so einstellen, dass er fortan eine geöffnete Verbindung anzeigt. Dann kann es sein, dass Sie in Ihrer Anwendung umgehend nachdem die Verbindung aufgebaut ist, ein bestimmtes Datum vom Server abrufen möchten. Sagen wir, Sie entwickeln gerade an einem Frontend für die Darstellung von Finanzinstrumenten in Realzeit. Nachdem Sie die WebSocket-Verbindung aufgebaut haben, wollen Sie immer die Daten für den DAX vom Server gestreamt haben. Daher setzen Sie unmittelbar eine erste Nachricht an den Server ab, die über ein proprietäres Protokoll die Daten des DAX anfragt. Hier soll das Protokoll durch das `Add`-Kommando ein Finanzinstrument der Beobachtungsliste des Clients hinzufügen. Angegeben wird das Wertpapier durch die *International Securities Identification Number* (ISIN), die über eine zwölfstellige Buchstaben-Zahlen-Kombination eine Identifikation für ein Wertpapier darstellt.

Fehler können immer auftreten. Bei der Programmierung verteilter Anwendungen trifft dies im besonderen Maße zu, da es eine Vielzahl von Fehlerquellen gibt, auf die Sie manchmal nicht einmal mit Ihrem eigenen Code Einfluss nehmen können. Denken Sie z. B. dabei an die Netzwerkverbindung. Hier bleibt Ihnen also keine andere Wahl, als mit diesem Fehler umzugehen.

Listing 5.3 Ausgabe eines Fehlercodes und der Fehlerbegründung in der Konsole

```
1 ws.onerror = function(event) {
2     var reason = event.reason;
3     var code = event.code;
4     console.log('Ein WebSocket-Fehler ist aufgetreten: ' +
          reason + '(' + code + ')');
5 };
```

Im Beispiel wird auf den Fehlercode und die formulierte Fehlerbegründung zugegriffen, die zum Fehlerfall geführt haben, und schlicht in der Konsole ausgegeben. Mehr zu den definierten Fehlercodes und Fehlerbegründungen haben wir in Abschnitt 4.7.3 besprochen.

Der Event Handler `onmessage` wird immer dann aktiviert, wenn eine Nachricht eingeht. Wenn Sie z. B. vorhaben, Ihre auf HTML5 basierenden Präsentationsfolien via WebSockets vor- und zurückzublättern, dann könnten Sie das durch das Absetzen von `next`- bzw. `prev`-Kommandos. Sie müssen dann im `onmessage`-Event-Handler Ihres WebSocket-Objekts eine Funktion registrieren, die eingehende Nachrichten nach den entsprechenden Kommandos absucht (siehe Listing 5.4).

Listing 5.4 Registrieren einer Funktion im `onmessage`-Event-Handler

```
ws.onmessage = function(message) {
    var data = message.data;
    if(data == 'next') {
        nextSlide();
    }
    else if(data == 'prev') {
        prevSlide();
    }
    else {
        // Ignoriere unbekannte Kommandos
    }
};
```

Ein WebSocket-Objekt kann über zwei Wege in den CLOSED-Zustand gelangen (siehe Abschnitt 5.4 und Bild 5.1). In beiden Fällen wird der `onclose`-Event-Handler aktiviert. Haben Sie dem entsprechenden Attribut Ihres WebSocket-Objekts eine Funktion zugewiesen, kommt diese dann zur Ausführung (siehe Listing 5.5).

Listing 5.5 Registrieren einer Funktion im `onclose` Event Handler

```
ws.onclose = function(event) {
    if(this.readyState == 2) {
        console.log('Schliesse die Verbindung...');
        console.log('Die Verbindung durchlaeuft den Closing-
            Handshake');
    }
    else if(this.readyState == 3) {
        console.log('Verbindung geschlossen...');
        console.log('Die Verbindung wurde geschlossen oder konnte
            nicht aufgebaut werden.');
    }
    else {
        console.log('Verbindung geschlossen...');
        console.log('Nicht behandelter readyState: ' +
            this.readyState);
    }
};
```

Alternativ können Event Handler auch mit der `addEventListener()`-Methode definiert werden. In Anlehnung an das zuvor in Listing 5.2 angegebene Beispiel zum `onopen`-Event Handler können Sie das gleiche Resultat mit dem Codeschnipsel aus Listing 5.6 erzielen.

Listing 5.6 Definition eines Event Handlers mit der `addEventListener()`-Methode

```
ws.addEventListener("open", function() {
    console.log('Die WebSocket-Verbindung wurde aufgebaut.');
    turnConnectionIndicatorOn();
    ws.send('Add: ISIN=DE0008469008');
};
```

Diese alternative Definition von Event Handler-Funktionen hat dann Vorteile, wenn Sie einem Event mehr als *einen* Handler zuweisen wollen. Dies können Sie nur mit der Verwendung der `addEventlistener()`-Methode realisieren.

5.7 Erstes vollständiges Prögrämmchen

So, an dieser Stelle wissen Sie schon genug, um einen ersten WebSocket-Client gemäß der W3C-API in JavaScript implementieren zu können. Probieren Sie es doch einfach mal aus! Lesen Sie nicht weiter und versuchen Sie ein vollständiges JavaScript-Programm anzugeben, das eine WebSocket-Verbindung zum Echo-Server aufbaut, der unter der URL

ws://echo.websocket.org/

verfügbar ist. Wenn die Verbindung aufgebaut werden konnte, soll Ihr Programm den String `'Hallo WebSocket Endpoint!'` an den Echo-Server schicken. Der Echo-Server würde einer derartigen Kommunikation mit einer einfachen Antwort mit dem gleichen Inhalt begegnen. Ihr Programm soll diese Antwort entgegennehmen und in der JavaScript-Konsole ausgeben. Danach sollen Sie die Verbindung wieder schließen. Falls es zu Fehlern kommt, sollen Sie diese mit Ihrem Programm nicht ignorieren, sondern vielmehr die Gründe dafür ebenfalls in die JavaScript-Konsole ausgeben. Also dann mal los!

Das nachfolgende Listing 5.7 zeigt eine mögliche Implementierung für den beschriebenen Echo-Client. Hierbei werden die Kernbestandteile der W3C WebSocket-API, die wir bisher besprochen haben, an einem Beispiel zusammenhängend dargestellt und verdeutlicht.

Listing 5.7 Vollständiger WebSocket-basierender Echo-Client

```
var ws = new WebSocket('ws://echo.websocket.org/');

ws.onopen = function() {
    console.log('WebSocket-Verbindung aufgebaut.');
    ws.send ('Hallo WebSocket Endpoint!');
    console.log('Uebertragene Nachricht: Hallo WebSocket Endpoint!');
};

ws.onmessage = function(message) {
```

```
10      console.log('Der Server sagt: ' + message.data);
11      ws.close();
12  };
13
14  ws.onclose = function(event) {
15      console.log('Der WebSocket wurde geschlossen oder konnte
            nicht aufgebaut werden.');
16  };
17
18  ws.onerror = function(event) {
19      console.log('Mit dem WebSocket ist etwas schiefgelaufen!');
20      console.log('Fehlermeldung: ' + event.reason +
            '(' + event.code + ')');
21  };
```

Schauen wir uns an, was im dargestellten Listing genau passiert. Zunächst haben wir ein WebSocket-Objekt instanziiert (Zeile 1), das bei der Objekterzeugung eine Verbindung zum Server aufbauen soll, dessen URL im Argument des Konstruktors angegeben ist. Das passiert dann auch, allerdings ohne die weitere Verarbeitung des Programms zu blockieren. Während also der Verbindungsaufbau innerhalb der Objekterzeugung noch im vollen Gange ist, wird der Rest des Programms asynchron (oder parallel) weiterverarbeitet. Dabei werden vier Funktionen den vier spezifizierten Event Handlern zugewiesen; jeweils eine Funktion, die angibt, was im Falle eines erfolgreichen Verbindungsaufbaus (`onopen`, Zeile 3), beim Eintreffen einer Nachricht vom Server (`onmessage`, Zeile 9), beim Eintreten eines Fehlers (`onerror`, Zeile 18) und bei dem Schließen der Verbindung (`onclose`, Zeile 14) ausgeführt werden soll. Wie es jetzt mit der Ausführung weitergeht, hängt davon ab, ob der Verbindungsaufbau zum Server erfolgreich ist oder nicht. Wenn Sie sich an das UML-Zustandsdiagramm in Bild 5.1 zurück erinnern, dann stehen wir jetzt genau am Scheideweg, der uns entweder zum OPEN- oder CLOSED-Zustand führt. Es sind also zwei Pfade durch das Programm möglich:

1. Schlägt der Verbindungsaufbau fehl, geht das Objekt `ws` in den CLOSED-Zustand über, wobei dies mit dem Absetzen eines close-Events einhergeht, was wiederum bewirkt, dass der `onclose`-Event Handler aktiviert wird. Dieser Programmdurchlauf erzeugt folglich nur eine einzige Ausgabe in der Konsole, und zwar die, die im `onclose` Event Handler angegeben ist: `'Der WebSocket wurde geschlossen oder konnte nicht aufgebaut werden.'`
2. Konnte der Verbindungsaufbau erfolgreich durchgeführt werden, geht das Objekt `ws` in den OPEN-Zustand über, wobei dies mit dem Absetzen eines OPEN-Events einhergeht, was wiederum bewirkt, dass der `onopen`-Event Handler aktiviert wird. Die für diesen Handler registrierte Funktion erzeugt zunächst eine informative Ausgabe in der Konsole über den erfolgreichen Verbindungsaufbau. Danach setzt sie eine Nachricht an den Server ab, die `'Hallo WebSocket Endpoint!'` zum Inhalt hat. Schließlich wird noch in der Konsole protokolliert, dass diese Nachricht an den Server abgeschickt wurde. Was als Nächstes ausgeführt wird, hängt davon ab, was passiert. Da es sich beim Server um einen Echo-Server handelt, wird dieser auf die an ihn gerichtete Nachricht mit einer Antwort reagieren, die exakt den gleichen Inhalt hat wie die des Clients. Es wird also eine Nachricht beim Client eintreffen, die einen message-Event auslöst, der wiederum den `onmessage`-Event Handler aktiviert. Die diesem Event Handler zugewiesene Funktion kommt zur Ausführung. Sie erzeugt eine Aussage in der Konsole, die

den Inhalt der Nachricht vom Server mit einem Präfix ausgibt, in diesem Beispiel `'Der Server sagt: Hallo WebSocket Endpoint!'`. Danach schließt der Client die Verbindung durch den Aufruf der `close()`-Methode. Dadurch wird ein close-Event abgesetzt, der den `onclose`-Event Handler aktiviert, wodurch die Ausgabe `'Der WebSocket wurde geschlossen oder konnte nicht aufgebaut werden.'` in der Konsole erscheint.

Da es den im Beispiel angeführten WebSocket-Echo-Server tatsächlich gibt, können Sie dieses Beispiel direkt ausprobieren, ohne sich an dieser Stelle schon Gedanken zur Implementierung der Serverseite machen zu müssen. Mit den in vielen Browsern enthaltenen JavaScript-Konsolen können Sie sich außerdem die beiden beschriebenen Ausführungspfade verdeutlichen. In Bild 5.2 geben wir Ihnen ein Beispiel.

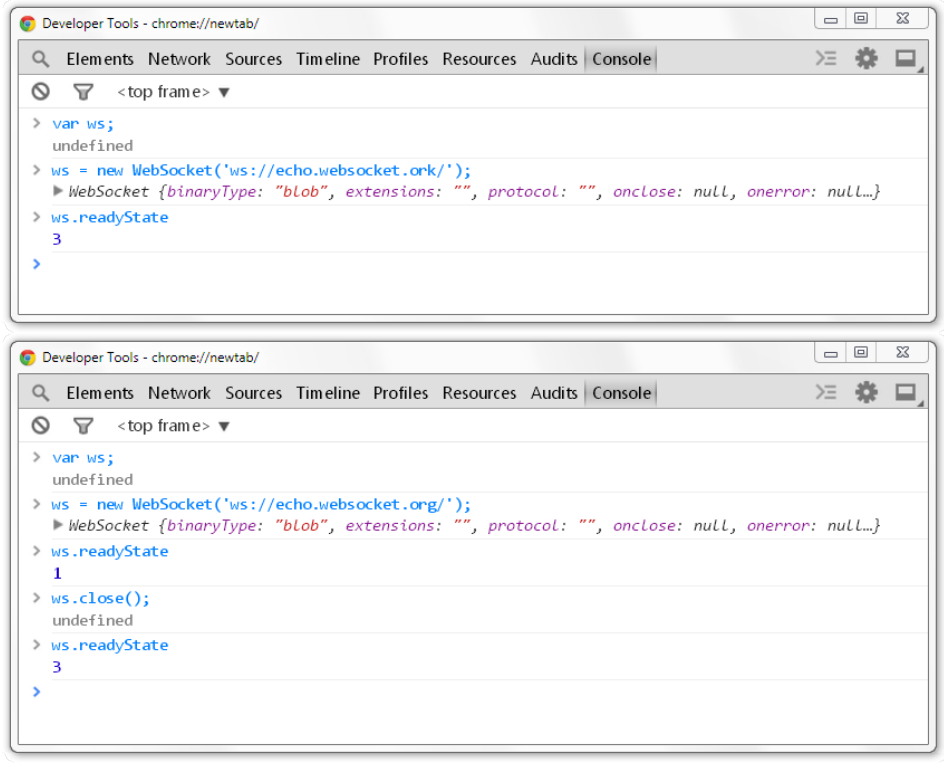

Bild 5.2 Zustandsübergänge bei der Objekterzeugung (JavaScript-Konsole von Chrome)

Im oberen Chrome-Browserfenster hat sich im Konstruktorargument ein Tippfehler eingeschlichen. Aus diesem Grund kann keine Verbindung zum angegebenen Server aufgebaut werden (dieser existiert so nicht). Daher ist das erzeugte WebSocket-Objekt anschließend im CLOSED-Zustand, was dem `readyState` mit dem Wert `3` entspricht. Im unteren Chrome-Fenster ist die URL im Konstruktorargument korrekt eingegeben. Eine Verbindung konnte folglich hergestellt werden und das WebSocket-Objekt befindet sich im OPEN-Zustand mit dem `readyState=1`. Das Aufrufen der `close()`-Methode beendet die Verbindung und versetzt das Objekt in den `readyState=3`.

5.8 Attribute

Ein WebSocket-Objekt verfügt neben dem `readyState` und den Event Handlern über weitere Attribute, auf die Sie zurückgreifen können, um in Ihren Programmen noch auf weitere Statusinformationen zur WebSocket-Verbindung zugreifen zu können. Diese haben wir aus dem W3C-Standard in die Tabelle 5.5 überführt.

Tabelle 5.5 Attribute eines WebSockets (ohne readyState und Event Handler) [Hic12b]

Attribute	Datentyp	Beschreibung
`binaryType`	DOMString	Gibt den Typ der Binärdaten an. Kann entweder den Wert `Blob` oder `ArrayBuffer` annehmen.
`bufferedAmount` (nur lesender Zugriff)	long	Gibt die Anzahl an Bytes an, die noch in der Warteschlange stehen und nicht versendet wurden. Dieser Wert wird nicht auf `0` gesetzt, wenn die Verbindung geschlossen wird. So können Sie überprüfen, ob alle abgesetzten Daten auch ihre Reise an das Ziel angetreten haben.
`extensions` (nur lesender Zugriff)	DOMString	Eine Liste mit den Erweiterungen, die vom Server ausgesucht wurden.
`protocol` (nur lesender Zugriff)	DOMString	Das aktuell benutzte Subprotokoll, das vom Server ausgewählt wurde.
`url` (nur lesender Zugriff)	DOMString	Die URL der Verbindung.

Das `url`-Attribut Ihres Objekts gibt die URL des WebSocket-Servers wieder, mit dem die Verbindung etabliert wurde. Wollen Sie sich diese in der Konsole ausgeben lassen, ist dies mit der folgenden Zeile getan:

```
console.log(
  'WebSocket-Verbindung aufgebaut mit ' + ws.url + '.');
```

Eine elegantere Lösung würde innerhalb des `onopen`-Event Handlers die URL des verbundenen Servers in eine in der Webseite dafür vorgesehene Statuszeile einfügen. Das Beispiel in Listing 5.8 zeigt das Prinzip, wobei angenommen wird, dass das HTML-Element mittels der eindeutigen ID `'server-status-line'` angesprochen werden kann. Das `url`-Attribut wird auf den Wert gesetzt, der bei der Objekterzeugung im Konstruktor angegeben wird. Der Wert verändert sich über die gesamte Lebensdauer des Objekts nicht mehr. Selbst im CLOSED-Zustand ist der Wert noch zugreifbar.

Listing 5.8 Beispiel für die Verwendung des `url`-Attributs innerhalb eines `onopen`-Event Handlers

```
1 ws.onopen = function() {
2     // Hier stehen nuetzliche initiale Einstellungen...
3     document.getElementById('server-status-line').innerHTML=this.url;
4 };
```

Wir haben das in der JavaScript-Konsole des Firebug für Sie nachvollziehbar gemacht (siehe Bild 5.3).

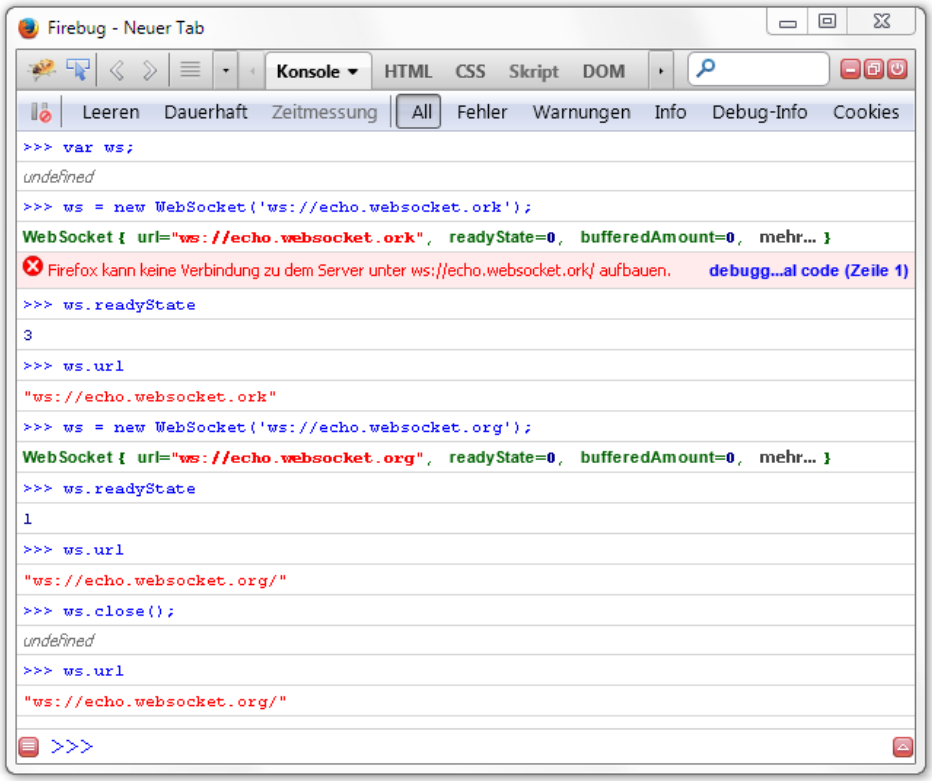

Bild 5.3 Lebensdauer des `url`-Attributs (JavaScript-Konsole von Firebug)

Zunächst erzeugen wir ein WebSocket-Objekt und damit eine WebSocket-Verbindung zu einem nicht existenten Server mit der falschen URL:

ws://echo.websocket.ork/

Daher geht das Objekt in den CLOSED-Zustand (`readyState=3`). In diesem Zustand ist das `url`-Attribut noch auf dem Wert der fälschlich eingegebenen URL. Im Anschluss erzeugen wir ein neues Objekt, diesmal mit der korrekten URL:

ws://echo.websocket.org/

Da die Verbindung zu diesem Server aufgebaut werden kann, befindet sich das Objekt im OPEN-Zustand (`readyState=1`). In diesem Zustand hat das `url`-Attribut als Wert die URL des verbundenen Servers inne. Schließen wir nun die Verbindung zum Server durch Aufrufen der `close()`-Methode, wechselt das Objekt in den CLOSED-Zustand (`readyState=3`). Wie schon zuvor beim Vertipper, bleibt der Wert des `url`-Attributs erhalten und zeigt die URL an, für die das Objekt erzeugt wurde.

Die Beobachtungen zur Lebensdauer von Attributwerten gelten auch für alle anderen Attribute aus Tabelle 5.5, für die wir im Folgenden noch jeweils ein Beispiel mit Ihnen durchgehen möchten.

Sie werden sich erinnern, dass wir in Abschnitt 4.2 darüber gesprochen haben, dass Sie beim Verbindungsaufbau in der Handshake-Anfrage ein oder mehrere Subprotokolle angeben können. Der Server sucht sich dann ein passendes aus. Ist kein passendes darunter, muss er die Verbindungsanfrage unterbinden. In Abschnitt 5.4 sind uns die Subprotokolle dann wieder begegnet. Hier konnten wir diese nämlich als optionale Parameter im Konstruktor angeben. Haben Sie Ihren Client nun so implementiert, dass er mit mehreren Subprotokollen umgehen kann und Sie diese dem Server zur Wahl stellen, müssen Sie wissen, für welches er sich entschieden hat. Das können Sie aus dem `protocol`-Attribut auslesen.

```
console.log(
    'Der WebSocket-Server nutzt das Subprotokoll ' + ws.protocol + '.');
```

Das Protokoll sieht von Haus aus Erweiterungen vor. Erweiterungen können entweder öffentlich gemacht werden oder sie bleiben eine reine private Angelegenheit. Im ersten Fall werden sie zu RFC-Standards. Im zweiten Fall bleiben sie proprietäre Lösungen. Aktuell gibt es noch keine standardisierten Erweiterungen. In den verschiedenen Vorläufer-Entwürfen zum RFC 6455 war eine Zeit lang eine Erweiterung mit dem Namen *deflate-stream* enthalten. Diese Erweiterung hat die Kompression der Daten zum Gegenstand, hat es aber aufgrund vieler konzeptioneller Schwächen schließlich nicht in den Standard geschafft. Außer einer Nennung in einem Beispiel ist nichts mehr von deflate-stream im RFC 6455 übrig geblieben. Zwei andere Erweiterungen befinden sich gerade innerhalb der IETF-Arbeitsgruppe HyBi als Drafts in Bearbeitung. Die eine Erweiterung trägt den Namen *WebSocket Per-frame Compression* [Yos12] und nähert sich dem Thema Datenkompression von einer anderen Seite. Zunächst legt sich diese Erweiterung nicht auf einen spezifischen Kompressionsalgorithmus fest. Dieser kann ausgehandelt werden. Ein einziger Algorithmus ist in der Erweiterung allerdings als kleinster Nenner vorgeschrieben und das ist der altbekannte Deflate-Algorithmus. Komprimiert werden dann die Nutzdaten in den Datenframes. Die andere Erweiterung mit dem Namen *Multiplexing Extension for WebSockets* [TY13] soll es ermöglichen, mehrere logische WebSocket-Kanäle über eine tatsächlich physisch vorhandene WebSocket-Verbindung zu schleusen. Mit dieser Erweiterung wird möglichen Skalierungsproblemen entgegengewirkt, die entstehen können, wenn ein Client viele unterschiedliche WebSocket-Verbindungen (z. B. in mehreren Tabs) zum gleichen WebSocket-Server aufgebaut hat. Da sich diese WebSocket-Erweiterungen noch stark im Fluss befinden, werden Sie in der überwiegenden Anzahl an Fällen einen leeren String im `extensions`-Attribut in den Headern der WebSocket-Handshakes vorfinden.

Es kann nicht immer alles prompt gesendet werden, so wie Sie es mit der `send()`-Methode absetzen. Die Gründe dafür können vielfältig sein. Der häufigste ist sicher eine begrenzte Datentransferkapazität. Um dies abzufedern, sieht der Browser einen Buffer vor, in dem noch nicht gesendete Daten eingereiht werden. Um herauszufinden, wie viele Bytes sich aktuell im Buffer befinden, steht das Attribut `bufferedAmount` bereit. Wenn wir beim Beispiel bleiben, dass eine Menge von Daten an den Server gesendet werden und diese nicht umgehend auf den Weg gebracht werden können, könnten Sie wie folgt mit der Situation umgehen, um Probleme durch unkontrollierte Überlastungen zu vermeiden.

In dem Codeschnipsel aus Listing 5.9 senden Sie in regelmäßigen Intervallen Mausbewegungsdaten an einen Server zur Unterstützung einer Usability-Studie. Alle 50 Millisekunden wird eine Funktion aufgerufen, die die Mauspositionsdaten ermittelt und an den Server schickt, dies aber nur tut, wenn keine „älteren“ Daten mehr im Buffer stehen und auf ihre Abreise warten. Mit dieser einfachen Methode passt sich die zeitliche Auflösung der Mausbewegungen automatisch an die Datentransferkapazitäten der Anwendung – in diesem Fall einer Usability-Testumgebung, die wir anhand eines vollständigen Beispiels in Abschnitt 8.3 besprechen – an.

Listing 5.9 Beispiel zur Vermeidung von unkontrollierten Überlastungen durch Datenmengen

```
var ws = new WebSocket('ws://usability.example.com/updates');
ws.onopen = function() {
    setInterval(function() {
        if(ws.bufferedAmount == 0)
        ws.send(getMousePositions());
    }, 50);
};
```

Wie wir uns im nachfolgenden Abschnitt 5.9 noch näher anschauen werden, sieht die WebSocket-API drei verschiedene Formen vor, wie binäre Daten an die `send()`-Methode übergeben werden können: entweder als `Blob` (Abkürzung für *Binary Large Object*), als `ArrayBuffer` oder als `ArrayBufferView`. Der letztgenannte Datentyp entspricht dabei im Wesentlichen einem `ArrayBuffer`, mit dem Unterschied, dass bei einem `ArrayBufferView` nur ein bestimmter Teil der vorliegenden Daten selektiert wird. Durch Auslesen des `binaryType`-Attributs können Sie feststellen, in welcher der beiden Formen die binären Daten eingetroffen sind. Initial steht der Wert auf `Blob`. Werden nur Strings übertragen, ändert sich der Wert des `binaryType`-Attributs nicht und verbleibt auf dem Initialwert (siehe Bild 5.4).

Werden Binärdaten empfangen, wird vom Browser auch auf Basis des `binaryType`-Attributs entschieden, ob auf die Daten als `Blob` oder `ArrayBuffer` zugegriffen werden kann. Genau diese zwei Werte dürfen Sie diesem Attribut zuweisen. Es kommen aber auch noch andere Faktoren bei der Entscheidungsfindung des Browsers zum Tragen. Darunter zählt z. B. die Datenmenge. Da ein ArrayBuffer im Hauptspeicher gehalten wird, sollte dieser nur bei kleineren Datenmengen verwendet werden. Größere Datenmengen sollten unbedingt als `Blob` temporär auf dem persistenten Speicher ausgelagert werden.

5.9 Datenübertragung

Zum Senden von Daten ist die `send()`-Methode im W3C-Standard definiert [Hic12b]. Wenn Sie sich den Standard genauer anschauen, werden Sie feststellen, dass sie vierfach überladen ist und die folgenden Argumente akzeptiert:

```
void send(DOMString data);
void send(Blob data);
void send(ArrayBuffer data);
```

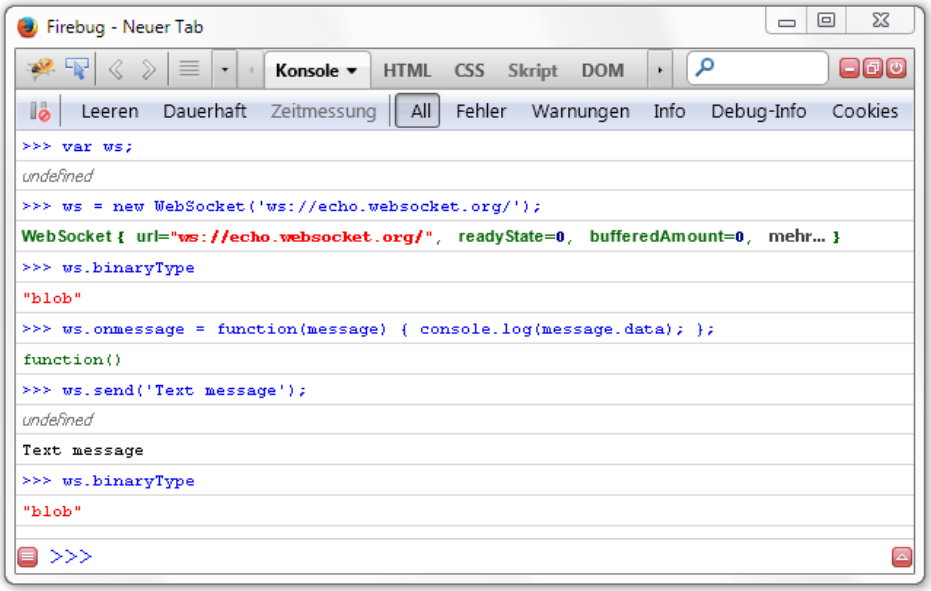

Bild 5.4 Das Attribut `binaryType` wird nur gesetzt, wenn binäre Daten empfangen werden.

```
void send(ArrayBufferView data);
```

Genau eine dieser vier `send()`-Methoden haben wir bereits häufiger in unseren Beispielen verwendet. Bisher haben wir jedes Mal die erstgenannte Alternative aufgerufen. Diese ist mit der textbasierten Übertragung einfacher Strings betraut. Die verbleibenden drei Varianten dienen allesamt der binären Datenübertragung, die wir bisher vernachlässigt haben. In den folgenden zwei Unterkapiteln wollen wir uns nun noch mal genauer mit den beiden bereitstehenden Übertragungsmodi auseinandersetzen.

5.9.1 Übertragung textbasierter Daten

Ist das Argument der `send()`-Methode ein String, wird er als solcher übertragen. Dazu wird der String in seine Einzelteile zerlegt und daraus eine Sequenz von UTF-8-Zeichen gebildet. Anschließend werden die UTF-8-Zeichen in einen WebSocket-Frame mit dem Text-Opcode eingefügt und gesendet. Der Aufruf

```
ws.send('a simple text message');
```

erzeugt auf der Leitung also folgenden WebSocket-Frame (siehe dazu auch Abschnitt 4.3.)

Bei der Übertragung dürfen Sie nicht außer Acht lassen, dass Ihre Daten evtl. zunächst in einen browser-internen Buffer gelangen und dadurch verzögert auf die Leitung kommen.

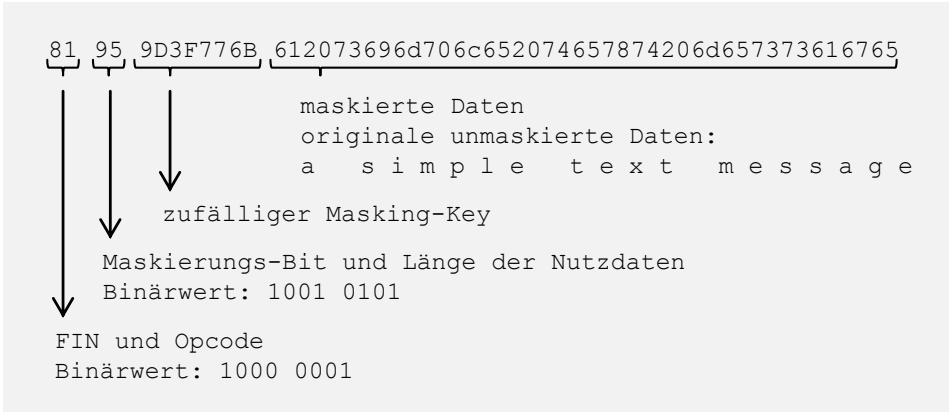

Bild 5.5 WebSocket-Frame mit Textdaten: „a simple text message“ (maskiert, da der Frame vom Client zum Server gesendet wurde)

5.9.2 Übertragung binärer Daten

Mit binären Daten kann auf zwei verschiedene Arten umgegangen werden. Welche der beiden Möglichkeiten Sie bzw. der Browser wann verwenden, hängt über den Daumen gepeilt von der Datenmenge ab. Handelt es sich um viele Daten, ist das Verwenden eines BLOB ratsam. Dies können Sie quasi schon aus dem Akronym ablesen (*Binary Large Object*). Anderenfalls stehen Ihnen `ArrayBuffer`- oder `ArrayBufferView`-Objekte zur Verfügung. Kleinere Datenstrukturen wie z. B. Array-Variablen, die ohnehin im Hauptspeicher Ihrer Anwendung gehalten werden, können Sie über `ArrayBuffer` abwickeln. `Blob`-Objekte sollten Sie immer dann verwenden, wenn die Daten potenziell groß werden können. Beispiele hierfür sind Dateiinhalte.

Im W3C-Standard findet sich eine ähnliche Empfehlung für die Browserhersteller wieder. Auf Basis dieser Empfehlungen sollen von den Browsern entsprechende Strategien entwickelt werden, wie mit den binären Daten umgegangen werden soll. Kleinere Datenmengen können im Hauptspeicher verbleiben. Größere Datenmenge sollten als Ganzes auf die Festplatte ausgelagert werden, um den Hauptspeicher nicht über die Maße zu beanspruchen. Wo genau die Grenzen liegen und wie genau damit umgegangen werden soll, schreibt der Standard nicht vor. Diesen Freiheitsgrad sollten Sie im Hinterkopf behalten, da dieser zu unterschiedlichen Implementierungen und damit auch zu unterschiedlichem Verhalten der Browser führt. Sie sollten daher immer prüfen, in welcher Form Ihnen die Binärdaten bereitgestellt werden, bevor Sie auf die empfangenen Daten zugreifen. Auch wenn Sie zuvor durch das Setzen des `binaryType`-Attributs auf den Wert `ArrayBuffer` den Wunsch ausgedrückt haben, die Daten als `ArrayBuffer` übergeben zu bekommen, heißt es nicht, dass Sie sie tatsächlich auch in dieser Form erhalten. Der Browser kann Ihre Präferenz überstimmen und sich für den `Blob`-Datentyp entscheiden. Eine Überprüfung auf den Binärdatentyp hat folgenden prinzipiellen Aufbau (siehe Listing 5.10):

Listing 5.10 Prinzipieller Aufbau einer Überprüfung auf den Binärdatentyp

```
1 ws.onmessage = function(event) {
2     if(event.data instanceof ArrayBuffer) {
```

```
        // Verarbeitung der ArrayBuffer-Daten
    }
    else if(event.data instanceof Blob) {
        // Verarbeitung der BLOB-Daten
    }
    else {
        onlose.log("Fehler: Es werden nur Binaerdaten
            unterstuetzt!");
    }
}
```

Ist das Argument der `send()`-Methode ein `Blob`-Objekt, werden die rohen Bytes in einen WebSocket-Frame mit dem Binary Frame Opcode versehen und verschickt. Das folgende Beispiel zeigt, wie Sie eine Datei vom lokalen Dateisystem via WebSockets als `Blob` an den Server schicken können (siehe Listing 5.11):

Listing 5.11 Verschicken einer Datei als BLOB-Objekt

```
var file = document.querySelector('input[type="file"]').files[0];
ws.send(file);
```

Für große Dateien hat ein derartiges Vorgehen wie gesagt den Vorteil, dass die Daten vom Browser nicht vollständig in den Hauptspeicher geladen werden müssen. Stattdessen werden die Daten von der Festplatte häppchenweise eingelesen und an den Empfänger gestreamt. Dieser wiederum kann auch die Tatsache nutzen, dass es sich um einen BLOB handelt, und die darüber bereitgestellten Daten ebenfalls direkt auf die Festplatte schreiben; auch hier wieder aus dem Grund, nicht alle Daten in den Hauptspeicher schreiben zu müssen.

Die verbleibenden zwei Varianten funktionieren äquivalent, mit dem Unterschied, dass geringe Datenmengen als `ArrayBuffer`- oder `ArrayBufferView`-Objekt der `send()`-Methode als Argument übergeben werden. Das folgende Beispiel zeigt, wie Sie die Pixelinformationen aus einem `canvas` entnehmen, daraus ein `ArrayBuffer`-Objekt erzeugen und dieses dann an den Server schicken können:

Listing 5.12 Verschicken von Pixelinformationen eines Canvas als ArrayBuffer-Objekt

```
var img = canvas_context.getImageData(0, 0, 400, 320);
var binary = new Uint8Array(img.data.length);
for (var i=0; i<img.data.length; i++) {
    binary[i] = img.data[i];
}
ws.send(binary.buffer);
```

Die Pixelinformation wird von der `getImageData()`-Methode als `ImageData`-Objekt zurückgegeben. Das `ImageData`-Objekt wiederum verwaltet die Pixel in einem eindimensionalen Array mit dem Namen `data`. Die Reihenfolge der Pixel geht von links nach rechts und von Zeile zu Zeile, beginnend in der linken oberen Ecke. Jedes Pixel wird dabei durch vier Elemente im Array repräsentiert, die die Farbe durch den Anteil an Rot, Grün und Blau sowie die Deckkraft (Alphakanal) bestimmen. Jeder Wert muss dabei im Bereich von 0 bis 255 liegen.

5.10 Verbindungsabbau

Beim Schließen des WebSocket-Kommunikationskanals muss unterschieden werden, wer von beiden Kommunikationspartnern die Verbindung abbaut. Ist es der Client selbst, so steht ihm dafür die `close()`-Methode bereit (siehe Abschnitt 5.10.1). Wird der Verbindungsabbau von der Serverseite initiiert, bekommt der Client dies durch das Auslösen des close-Events angezeigt (siehe Abschnitt 5.10.2).

5.10.1 Verbindung beenden

Zum Verbindungsabbau stellt die W3C-WebSocket-API dem Webbrowser eine `close()`-Methode bereit. Sie ist wie folgt anhand der WebIDL [McC12] spezifiziert:

```
void close([Clamp] optional unsigned short code,
                   optional DOMString reason);
```

Die `close()`-Methode hat keinen Rückgabewert und kann parameterlos aufgerufen werden. Dies bewirkt, dass der Browser einen Close-Frame an den Server schickt, der den Statuscode `1000` mit der Bedeutung `Normal Closure` enthält (siehe Tabelle 4.1).

Entstehen in Ihren Programmen im Browser unerwartete Situationen und müssen Sie dadurch die Verbindung abbrechen, können Sie der `close()`-Methode zusätzliche Parameter übergeben, mit denen Sie der Gegenseite signalisieren können, was zum vorzeitigen Abbruch der Verbindung geführt hat. Kombinationsmöglichkeiten, die sich aus der WebIDL-Spezifikation der Methode ergeben, sind die folgenden:

```
void close(unsigned short code);
void close(unsigned short code, DOMString reason);
```

Mit dem ganzzahligen Parameter `code` können Sie einen Statuscode angeben, der sich vom Standardcode `1000` unterscheidet. Da der Parameter in der Spezifikation mit `[Clamp]` markiert ist, sind nicht alle Zahlenwerte des Datentyps als Werte zulässig. Mögliche Werte sind weiterhin die `1000` und die Zahlen von `3000` bis `4999`. Genau in diesem Bereich befinden sich gemäß RFC 6455 die Statuscodes, die für Frameworks, Libraries und Anwendungen vorgesehen sind, und zwar die standardisierten Codes im Bereich von `3000` bis `3999` und die Codes von `4000` bis `4999` für proprietär festgelegte Codes (siehe Abschnitt 4.7.3). Geben Sie einen anderen Zahlenwert als ersten Parameter ein, wird eine `InvalidAccessError`-Exception ausgelöst.

Mit dem String `reason` können Sie zusätzliche Informationen zum Verbindungsabbruch an den Server senden. Der String ist allerdings längenbeschränkt. Im W3C-Standard ist definiert, dass dieser maximal 123 Bytes betragen darf. Wird dieses Längenlimit überschritten, wird eine `SyntaxError`-Exception ausgelöst.

5.10.2 Close-Event

Eine Verbindung kann ordnungsgemäß, d. h. durch das Senden eines Close-Frames vom Server oder Client, geschlossen werden. Wenn der Server die Verbindung trennt, wird dies

der Anwendung im Browser durch einen entsprechenden `close`-Event signalisiert. Dieser stößt die Funktionen an, die sich am `onclose`-Event-Handler registriert haben. Dem `onclose`-Handler wird ein `event`-Objekt als Parameter mitgegeben, das die in Tabelle 5.6 aufgelisteten Attribute enthält. Diese entsprechen im Großen und Ganzen den Informationen, die im empfangenen Close-Frame enthalten sind.

Tabelle 5.6 Close-Event Attribute [Moz14]

Attribut	Datentyp	Beschreibung
`code`	long	Gibt den Close-Code wieder, der durch den Server bestimmt wird.
`reason`	DOMString	Eine Textnachricht, die den Grund für die Beendigung angibt. Diese Nachricht ist von Servern bzw. Subprotokollen selbst bestimmbar.
`wasClean`	boolean	Gibt an, ob eine Verbindung ordnungsgemäß geschlossen wurde (`wasClean=true`) oder nicht (`wasClean=false`).

Das `wasClean`-Attribut gibt über einen Boolean-Wert an, ob die Verbindung ordnungsgemäß geschlossen wurde oder nicht. Eine Verbindung wird als ordnungsgemäß geschlossen angesehen, wenn sowohl Client als auch Server jeweils einen Close-Frame gesendet und empfangen haben. Der Grund für die Beendigung der Verbindung kann aus dem `code`-Attribut ermittelt werden. Bei einer ordnungsgemäßen Trennung beinhaltet es den Statuscode `1000`. Alle in RFC 6455 standardisierten Statuscodes sind in Tabelle 4.1 in Abschnitt 4.7.3 aufgelistet.

Das folgende Programmfragment illustriert, wie Sie auf diese Attribute zugreifen und diese in der JavaScript-Konsole Ihres Webbrowsers ausgeben können:

Listing 5.13 Beispiel für die Verwendung der close-Event-Attribute

```
1 ws.onclose = function(event) {
2     if(!event.wasClean) {
3         console.log('WS-Fehlercode: ' + event.code);
4         console.log('WS-Fehlerursache: ' + event.reason);
5     }
6 }
```

5.11 Ausblick: HTTP 2.0/SPDY

Das World Wide Web hat sich seit der letzten Standardisierung der Protokollversion HTTP 1.1 im Jahre 1999 sehr stark verändert. Heutzutage sind Webseiten viel komplexer und enthalten deutlich mehr Inhalte, wie Bilder, Skripte oder CSS-Dateien. Auch die Dateigrößen der Inhalte sind deutlich höher als in den späten Neunzigerjahren des letztens Jahrtausends. Internetseiten werden heutzutage auch sehr oft an mobile Endgeräte wie Smartphones und Tablet-PCs ausgeliefert. Webseiten, die sehr große oder sehr viele Dateien enthalten, erzeugen mehr Traffic, wodurch für Benutzer mobiler Endgeräte höhere Kosten entste-

hen können. Dazu kann die Akkulaufzeit von Smartphones und Tablet-PCs durch größere Datenübertragungen verringert werden.

Durch diese gestiegenen Anforderungen werden Nachteile des heutigen HTTP-Protokolls deutlich sichtbar. Pro Anfrage kann nur eine Ressource zurückgeliefert werden. Des Weiteren können, wie wir schon besprochen haben, nur Daten angefordert werden, wenn der Client diese anfragt. Der Server ist nach aktuellem Stand nicht in der Lage, über das HTTP-Protokoll von sich aus Daten an den Client zu schicken.

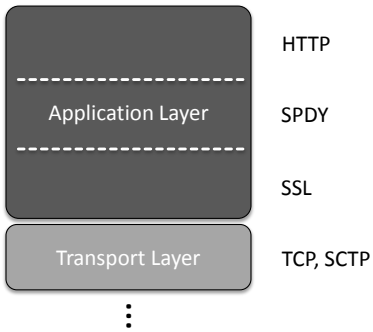

Bild 5.6 Einordnung von SPDY in das OSI-Modell bezogen auf das Internet

Um das Internet zu beschleunigen und das HTTP-Protokoll an den aktuellen Stand der Technik anzupassen, hat Google SPDY (ausgesprochen „Speedy“) [The10] entwickelt. Mit dieser Technologie soll u. a. die Performance der Ladezeiten verbessert und der Traffic der Übertragung verringert werden. Google hat sich das Ziel gesetzt, mit SPDY die Ladezeit von Webseiten um bis zu 50% zu verringern. SPDY benutzt TCP als Transportschicht, somit müssen Entwickler ihre Netzwerkinfrastruktur nicht ändern. Die einzige Voraussetzung besteht darin, dass der Client und der Server das SPDY-Protokoll unterstützen. Unterstützt der Client kein SPDY, so wird wieder auf das normale HTTP zurückgegriffen. Zusätzlich soll SPDY direkt über SSL verschlüsselt werden, um das Mitschneiden von Traffic nutzlos zu machen.

Damit die oben genannten Vorteile umgesetzt werden können, erlaubt das SPDY-Protokoll, mehrere HTTP-Anfragen in einer TCP-Verbindung zu stellen. Um Ladezeiten zu verringern, werden die Header bei der Übertragung zusätzlich komprimiert. SPDY ermöglicht auch die Priorisierung von Anfragen, sodass wichtige Daten früher ankommen können als weniger wichtige Daten. Ein weiteres Feature des Google-Protokolls ermöglicht dem Server, Daten eigenständig zu „pushen“, ohne dass der Client vorher eine Anfragen stellen musste. Der Server hat auch die Möglichkeit, dem Client wichtige Hinweise zu geben, z. B. welche Daten er noch laden muss, von denen er zum aktuellen Zeitpunkt noch nichts wusste.

Google geht sogar einen Schritt weiter und bietet ein neues Protokoll namens *Stream Control Transmission Protocol* (SCTP) [SXM+00] als Transport Layer für SPDY an, das TCP irgendwann ersetzen soll. SCTP bietet multiplexes Streamen und Überlastungskontrollen an.

SPDY ist bereits laut caniuse.com [Dev14b] ab Chrome 4, Firefox 13, Safari 8 und Opera 12.1 implementiert. Der Internet Explorer unterstützt das Protokoll ab der Version 11 aller-

dings nur unter Windows 8. Mit Safari kann diese Technologie noch nicht genutzt werden. Serverseitig ist SPDY schon weit verbreitet. Für die Webserver Apache, nginx und Node.js gibt es bereits Module und auch Jetty unterstützt SPDY seit den Versionen 7.6.2, 8.1.2 und 9.

Am 28. November 2012 hat die IETF den ersten Arbeitsentwurf zu HTTP 2.0 veröffentlicht, der auf Googles SPDY basiert.

Bei Google Chrome besteht die Möglichkeit, WebSockets über SPDY aufzubauen. Diese Implementierung ist aber nur experimentell und daher noch nicht stabil. Mit der Befehlszeile

```
--enable-websocket-over-spdy
```

starten Sie Google Chrome mit SPDY-Unterstützung für WebSockets. Der Startbefehl kann z. B. so aussehen:

```
C:\Users\SomeUser>"AppData\Local\Google\Chrome SxS\Application\
    chrome.exe" --enable-websocket-over-spdy
```