Bachelorarbeit im Studiengang B.Sc. Wirtschaftsinformatik

Realisierung einer bidirektionalen Chat-Anwendung auf Basis von HTML5 WebSockets mit Java und Angular 4 zum Einsatz im seminaristischen Kontext

Implementation of a bidirectional chat application utilizing HTML5 WebSockets in Java and Angular 4 for use in a seminaristic context

Andre Weinkötz

15. Februar 2018

Fakultät für Informatik und Mathematik
Hochschule München

Betreuer: Prof. Dr. Mandl

Abstract

Hier steht am Ende was genau eigentlich mit dem Unsinn hier erreicht werden soll.

Inhaltsverzeichnis

Abstract 2

1 Einleitung 5

2 Das WebSocket Protokoll 6

2.1 WebSocket-Frames . . . 6

2.1.1 Header-Felder . . . 7

2.1.2 Non-Control-Frames . . . 8

2.1.3 Control-Frames . . . 10

2.2 WebSocket Kommunikationsphasen . . . 12

2.2.1 Verbindungsaufbau . . . 13

2.2.2 Datentransfer . . . 15

2.2.3 Verbindungsabbau . . . 15

2.3 Alternative Verfahren . . . 16

2.3.1 Polling . . . 17

2.3.2 Long-Polling . . . 17

2.3.3 Comet . . . 17

2.3.4 Server-Sent Events . . . 19

3 Die WebSocket-API 20

3.1 Elemente der W3C WebSocket-API . . . 20

3.1.1 WebSocket Lebenszyklus . . . 20

3.1.2 Event-Handling . . . 22

3.1.3 Kommunikation . . . 22

3.2 Die Java™ API for WebSocket (JSR-356) . . . 23

3.2.1 Grundlagen . . . 24

3.2.2 Kommunikation . . . 27

3.2.3 En- und Decoding . . . 29

3.2.4 Konfiguration und `Session` . . . 31

3.2.5 Path-Mapping . . . 34

4 Fachliche Anforderungen 35

4.1 Analyse des DaKo-Frameworks . . . 35

4.2 Anforderungen an WebSocket-Chat . . . 35

5 Technische Anforderungen 35

5.1 Entwurf des WebSocket-Chats . . . 35

5.2 Implementierung des WebSocket-Chats . . . 35

6 Evaluation 35

6.1 Test-Umgebung . . . 35

6.2 Leistungsanalyse WebSocket-Chat 35

7 Zusammenfassung und Ausblick **35**

7.1 Einsatzgebiete 35

1 Einleitung

Die Anforderungen an Webanwendungen haben sich in den vergangenen Jahren stark verändert. Mobile Geräte wie Smartphones oder Tablets ersetzen stationäre Systeme, Webanwendungen sollen zur Kollaboration eingesetzt werden und Buchungssysteme oder Finanzanwendungen verlangen die Bearbeitung tausender Anfragen mit minimaler Verzögerung. Das Hypertext Transfer Protocol (HTTP) bietet hier keine zufriedenstellende Lösung. Um den Anforderungen an moderne Webanwendungen gerecht zu werden, spezifizierten das World Wide Web Consortium (W3C) und die Internet Engineering Taskforce (IETF) das WebSocket-Protokoll mit zugehöriger JavaScript-API.

Die Entwickler des WebSocket Protokolls machten es sich zur Aufgabe, einen Mechanismus zu schaffen, der es browserbasierten Anwendungen ermöglicht bidirektional zu kommunizieren ohne dabei auf mehrere HTTP Verbindungen zu öffnen [FM11]. Dabei sollte auf zusätzliche Methoden - wie beispielsweise den Einsatz von XMLHttpRequests, iframes oder long polling - verzichtet werden.

2 Das WebSocket Protokoll

Die Arbeit an der Spezifikation des WebSocket Protokolls begann bereits 2009, als es von Google in Zusammenarbeit mit Apple, Microsoft und Mozilla im Rahmen ihrer Kooperation WHATWG[1] der Internet Engineering Task Force (IETF) vorgeschlagen wurde. Bis Mai 2010 wurden noch zahlreiche Verbesserungen hinzugefügt, bis es schließlich in Version 76 [Hic10] im August 2010 an die BiDirectional or Server-Initiated HTTP (HyBi) Task Force der IETF zur Weiterentwicklung übergeben wurde [Fet10]. Neben vielen anderen Fortschritten wurde das Protkoll um die Möglichkeit erweitert binäre Dateien auszutauschen, sowie Sicherheitslücken geschlossen, die in Verbindung mit Proxy-Servern entstehen konnten. Im Dezember 2010 wurde das WebSocket Protokoll von der IETF zu dem Request for Comment (RFC) 6455 erklärt [FM11]. Dieser definiert auf über 70 Seiten neben den Kommunikationsphasen und verschiedenen Frames der WebSockets noch zahlreiche weitere technische Details und Defintionen. In diesem Kapitel sollen die wichtigsten Grundlagen der Protokollspezifikation dargelegt werden.

2.1 WebSocket-Frames

Die Spezifikation eines WebSocket-Frames sieht eine Zweiteilung vor, wie sie bei Protokollnachrichten üblicherweise vorgenommen wird. Ein WebSocket-Frame besteht demnach aus einem Header Bereich der Kontrolldaten enthält, sowie dem Payload, welcher die Nutzdaten beinhaltet. Anhand des Aufbaus ist leicht erkennbar, dass die Kommunikation über WebSockets mit

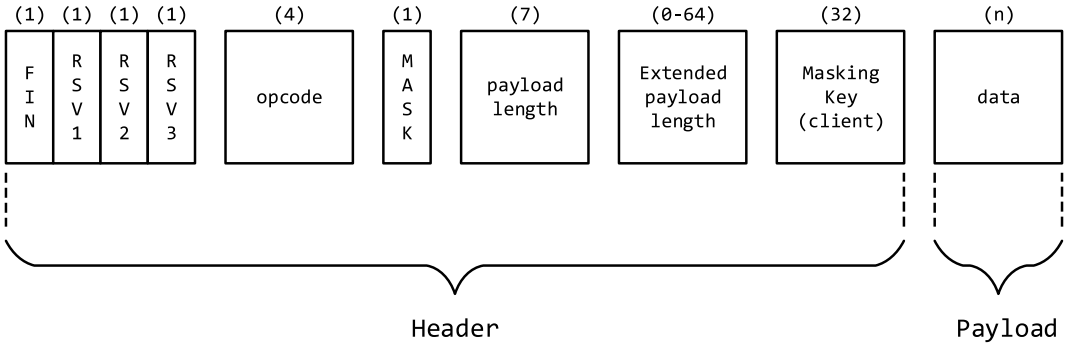

Abbildung 1: Vereinfachte Darstellung eines WebSocket Frames

deutlich geringeren Paketgrößen arbeiten kann als HTTP. Eine Nachricht mit einfacher Längenangabe, die von Server zu Client geschickt wird, erzeugt durch ihren Header lediglich einen Overhead von zwei Byte. Da WebSocket-Frames auf umgekehrtem Weg zusätzlich noch maskiert werden müssen, entspricht die minimale Länge eines clientseitigen Frames sechs Byte. Im Vergleich dazu beträgt der Overhead bei einer einfachen GET-Anfrage über HTTP/1.1 beim Aufruf des Hosts unter cs.hm.edu 35 Byte. Die Angabe des Hostnamens in HTTP/1.1 ist obligatorisch und trägt maßgeblich zur Größe des HTTP-Frames bei [Lea+99, S. 128].

[1] Web Hypertext Application Technology Working Group unter: www.whatwg.org

2.1.1 Header-Felder

Im nachfolgenden Abschnitt werden die Felder sowie deren jeweilige Aufgabe detailliert erläutert:

FIN (1 Bit) Das FIN-Flag gibt an, ob es sich bei dem empfangenen Frame um ein Fragment handelt. Kann der Payload nicht auf einmal übertragen werden, so wird die Übertragung über mehrere Frames gesteuert. Das FIN-Flag aller unvollständigen Fragmente weist den Wert 0 auf, wohingegen das finale Fragment mit einer 1 gekennzeichnet ist. Wenn alle Nutzdaten mit einem einzigen Frame übertragen werden können, so trägt dessen FIN-Flag ebenfalls den Wert 1.

RSV1, RSV2, RSV3 (jeweils 1 Bit) Diese Flags sind für zukünftige Implementierung reserviert und finden aktuell keine standardisierte Anwendung. Die Ausnahme stellt das RSV1-Bit dar, welches verwendet wird um anzuzeigen, ob eine Nachricht komprimiert wurde oder nicht [Int11]. Daher tragen sie meist den Wert 0. Ist eines dieser Flags gesetzt und die empfangende Stelle hat keine Erweiterung, welches das jeweilige Flag interpretieren kann, so muss die WebSocket-Verbindung geschlossen werden [FM11, S. 27].

opcode (4 Bit) Das opcode-Feld enthält die Angabe wie der Payload zu interpretieren ist. Neben den sogenannten Datenframes bzw. Non-Control-Frames, welche die Art der übertragenen Daten kennzeichnen, sind Control-Frames definiert, die zur Steuerung und Überprüfung der Verbindung verwendet werden. Wird ein unbekannter Opcode empfangen, so wird die WebSocket-Verbindung geschlossen.

- **0x0** definiert ein Fortsetzungs-Frame
- **0x1** definiert ein Text-Frame
- **0x2** definiert ein Binary-Frame
- **0x3-7** reserviert für weitere Non-Control-Frames
- **0x8** definiert ein Verbindung beenden-Frame
- **0x9** definiert ein Ping-Frame
- **0xA** definiert ein Pong-Frame
- **0xB-F** reserviert für weitere Control-Frames

MASK (1 Bit) Ist das MASK-Flag gesetzt, so ist der Payload maskiert. Frames, die clientseitig initiiert wurden, müssen maskiert sein, wohingegen serverseitige Frames keine Maskierung vornehmen dürfen. Der verwendete Masking-Key muss im gleichnamigen Header-Feld hinterlegt werden.

payload length (7 Bit) Hier wird die Länge der Nutzdaten angegeben. Sind diese nicht größer als 125 Bytes, so können sie mit nur einem Frame übertragen werden. Nutzdaten mit einem höheren Speicherplatzbedarf werden je nach Größe mit dem Wert 126 bzw. 127 angegeben. Diese Angaben führen zur Verwendung des Header-Felds *Extended payload length*.

Extended payload length (16 Bit oder 64 Bit) Überschreitet die Länge des Payloads 125 Byte, so wird dieses Feld verwendet. Je nach Wert im Header-Feld *payload length* umfasst die *Extended payload length* zwei Byte oder acht Byte. Dementsprechend beträgt die maximale Länge der Nutzdaten in einem WebSocket-Frame $2^{64} - 1$ Byte [GLN15, S. 41]

Masking Key (0 oder 32 Bit) Ein Frame, der von einem Client an einen Server gesendet wird, muss mit einem 32-Bit Schlüssel maskiert werden. Dieser wird vom Client generiert und im gleichnamigen Header-Feld abgelegt. Nachrichten, die von einem Server an einen Client gesendet werden, enthalten dieses Feld nicht.

data (n Byte) Hier befinden sich die eigentlichen Nutzdaten. Ihre Größe wird in den entsprechenden Längenfeldern des Headers hinterlegt. Werden lediglich Steuerungsinformationen übertragen, wird kein Payload mitgesendet.

2.1.2 Non-Control-Frames

Durch das Header-Feld *opcode* wird die Art der Nutzdaten angegeben. Es können sowohl textbasierte Daten als auch Binärdaten übertragen werden. Bei der Kommunikation über WebSockets gibt es hier einige zusätzliche Mechanismen, die für den fehlerfreien Ablauf einer Übertragung notwendig sind. Der folgende Abschnitt handelt von der bereits aus anderen Protokollen bekannten Fragmentierung, die WebSocket-Frames zur sequentiellen Übertragung unvollständiger Daten nutzt sowie der aufgrund von Sicherheitsproblemen eingeführten Maskierung clientseitig initierter Nachrichten.

Fragmentierung

In Abschnitt 5.4. des RFC6455 ist die Fragmentierung von WebSocket-Frames beschrieben. Primäre Anwendung erfährt dieses Verfahren, wenn die Daten zum Zeitpunkt der Übertragung noch nicht vollständig vorliegen oder nicht zwischengespeichert werden können. Wird die vom Server oder einer vermittelnden Stelle festgelegte Puffergröße überschritten, so wird ein Nachrichtenfragment mit dem Inhalt des Puffers gesendet [FM11, S. 32].

Folgende Grundsätze müssen bei der Fragmentierung beachtet werden. Erweiterungen werden hierbei außer Acht gelassen:

- Unfragmentierte Nachrichten werden mit einem einzelnen Frame übertragen (FIN-Bit = 1, opcode $\neq 0$)
- Die Übertragung fragmentierter Nachrichten erfolgt über mindestens einen Frame mit FIN-Bit = 0 und opcode $\neq 0$ und wird mit einem Frame FIN-Bit = 1 und opcode $\neq 0$ abgeschlossen. Die Summe der Payloads aller Fragmente entspricht dem Payload eines Einzelframes bei entsprechender Puffergröße.
- Control-Frames (Abschnitt 2.1.3) dürfen nicht fragmentiert werden. Sie können zwischen der Übertragung einer fragmentierten Nachricht gesendet werden, um beispielsweise die

Latenzzeit eines Pings möglichst gering zu halten. Die Verarbeitung zwischengesendeter Control-Frames muss von allen Endpunkten unterstützt werden.

- Der Empfänger muss die Fragemente einer Nachricht in der gleichen Reihenfolge erhalten, in welcher sie vom Sender aufgegeben wurden.
- Vermittler dürfen an der Fragmentierung keine Änderungen vornehmen, falls eines der RSV-Bits gesetzt ist und dessen Bedeutung dem Vermittler nicht bekannt ist.

Daraus folgt, dass die Verarbeitung sowohl fragmentierter als auch unfragmentierter Nachrichten von allen Endpunkten unterstützt werden muss. Da Control-Frames nicht fragmentiert werden dürfen, können Fragmente lediglich von den Typen Text, Binary oder einem der reservierten opcodes (0x3-7) sein. Die Angabe des Typs erfolgt durch den opcode des ersten Fragments. Alle weiteren Fragmente tragen den opcode 0x0, der dem Empfänger mitteilt, dass die empfangenen Nutzdaten an den Payload des vorangegangen Frames angehängt werden sollen [She17].

Maskierung

WebSocket Frames, von einem Client zu einem Server gesendet, müssen maskiert werden. Dieser Mechanismus war im ursprünglichen Draft für das WebSocket Protokoll nicht vorgesehen. In Folge einer Untersuchung durch eine Gruppe von Forschern rund um ein Team der Carnegie Mellon Universität wurden Sicherheitslücken in dem Konzept aufgedeckt. Bei der Verwendung transparenter Proxy-Server konnten sie durch deren fehlerhafte Implementierung des, bei dem Aufbau einer WebSocket Verbindung genutzten, *Upgrade* Mechanismus den Cache des Proxies infizieren [Hua+11]. Daraufhin wurde der Draft überarbeitet und um die Maskierung über eine bitweise XOR Verknüpfung ergänzt.

Da der zur Maskierung des Payloads verwendete Schlüssel im Frame enthalten ist, steht hier nicht die Vertraulichkeit der gesendeten Daten im Vordergrund. Vielmehr sollen Proxy-Server daran gehindert werden, den Inhalt des Payloads zu lesen. Somit wird verhindert, dass Angreifer den Payload manipulieren und diesen für einen Angriff gegen einen Proxy einsetzen können [GLN15]. Im Folgenden soll der Ablauf der Maskierung einer Textnachricht an einem Beispiel verdeutlicht werden:

Zunächst wählt der Client einen bisher nicht verwendeten 32 Bit Schlüssel aus, der zur Maskierung verwendet werden soll. Die Nachricht, die maskiert werden soll lautet "cs.hm.edu", konvertiert in hexadezimale Darstellung "63 73 2e 68 6d 2e 65 64 75". Als Schlüssel wird "3c 2e 3f 4a"verwendet. Dieser wird nun zyklisch auf den zu maskierenden Payload mit der XOR-Operation angewendet.

Die maskierte Nachricht wird mit dem Schlüssel in dem WebSocket-Frame abgelegt. Der Server kann diese nach dem Empfang durch erneute Anwendung der selbstinversen XOR-Operation demaskieren [GLN15].

Unmaskierte Bytes	63	73	2e	68	6d	2e	65	64	75
Operator	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Schlüssel	3c	2e	3f	4a	3c	2e	3f	4a	3c
Maskierte Bytes	5f	5d	11	22	51	00	5a	2e	49

Tabelle 1: Maskierung durch bitweise XOR-Verknüpfung

Maskierte Bytes	5f	5d	11	22	51	00	5a	2e	49
Operator	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Schlüssel	3c	2e	3f	4a	3c	2e	3f	4a	3c
Unmaskierte Bytes	63	73	2e	68	6d	2e	65	64	75
Nachricht	c	s	.	h	m	.	e	d	u

Tabelle 2: Demaskierung durch bitweise XOR-Verknüpfung

Die Übertragung von Server zu Client darf hingegen nicht maskiert werden. Empfängt ein Server eine unmaskierte bzw. ein Client eine maskierte Nachricht, so muss in beiden Fällen die Verbindung geschlossen werden. Dabei kann der im RFC6455 spezifierte Statuscode *1002 (protocol error)* beim Verbindungsabbau angegeben werden [FM11, S. 26].

2.1.3 Control-Frames

Control-Frames werden dazu genutzt, den Status einer WebSocket-Verbindung zu steuern und zu überwachen. Der RFC6455 unterscheidet dabei drei Arten von Control-Frames, welche anhand des *opcodes* unterschieden werden. Der Close-Frame wird zur Einleitung und Bestätigung einer schließenden Verbindung eingesetzt. Ping- bzw. Pong-Frames werden genutzt, um festzustellen ob der jeweilige Endpunkt noch erreichbar ist oder um eine Verbindung aktiv zu halten (keep-alive) [FM11, S. 35-36]. Im Folgenden werden die verschiedenen Arten der Control-Frames sowie deren Aufgaben erläutert.

Ping- und Pong-Frames

Wie eingangs erwähnt, dienen Ping- und Pong-Frames zur Verwaltung offener WebSocket-Verbindungen. Ping-Frames tragen den *opcode* 0x9, Pong-Frames hingegen den *opcode* 0xA. Das Übertragen von Nutzdaten bis zu 125 Byte durch einen Ping- oder Pong-Frame ist durch den RFC6455 zwar vorgesehen, kommt aber typischerweise nicht zum Einsatz [GLN15, S. 48]. Enthält ein Ping-Frame Nutzdaten, so müssen diese von einem Pong-Frame übernommen und zurückgesendet werden. Empfängt ein Endpunkt einen Ping-Frame, so muss schnellstmöglich mit einem Pong-Frame darauf reagiert werden. Werden mehrere Ping-Frames empfangen, bevor ein Pong-Frame zurückgesendet wurde, so kann lediglich der zuletzt eingetroffene Ping-Frame beantwortet werden [FM11, S. 36]. Ping- und Pong-Frames dienen aktuell als rein interner Mechanismus und können nicht über eine Schnittstelle direkt versendet werden [GLN15, S. 49].

	Ping-Frame	**Pong-Frame**
Byte	89 00	8a 80 (85 e1 ef 27)
Binär	1000 1001	1000 1010 (...)
Detail	FIN = 1, RSV n. gesetzt opcode = 0x9	FIN = 1, RSV n. gesetzt opcode = 0xA

Abbildung 2: Ping- und Pong-Frames zur Verbindungskontrolle

Bei den in Abbildung 2 dargestellten Frames handelt es sich um den Wireshark-Mitschnitt eines Heartbeats zwischen einem Client und dem Host unter *ws://echo.websocket.org*. Der Server sendet dabei einen Ping-Frame, um festzustellen, ob der Client noch erreichbar ist. Der Client antwortet mit einem korrespondierenden Pong-Frame. Da es sich um eine Nachricht von einem Client zu einem Server handelt, ist dieser Frame maskiert.

Close-Frames

Der Abbau einer WebSocket-Verbindung erfolgt durch den Versand eines Close-Frames. Der zugehörige *opcode* lautet 0x8. Empfängt ein Endpunkt einen Close-Frame, ohne bereits selbst einen solchen versendet zu haben, so muss er ebenfalls mit einem Close-Frame antworten. Wurde der initiale Close-Frame mit einem Status-Code versehen, so wird dieser in der Antwort übernommen [FM11, S. 35]. Wird ein Close-Frame empfangen, so können ausstehende Fragmente noch vor dessen Bestätigung gesendet werden. Allerdings kann dabei nicht garantiert werden, dass der Endpunkt, der den Verbindungsabbau eingeleitet hat, diese Nachrichten noch verarbeitet. Haben beide Endpunkte je einen Close-Frame gesendet und empfangen, so müssen sowohl die WebSocket- als auch die TCP-Verbindung getrennt werden. Der Server muss dies unverzüglich durchführen, wohingegen der Client auf den Server warten sollte [FM11, S. 36]. Sollte der Verbindungsabbau von beiden Endpunkten gleichzeitig eröffnet werden, so kann angenommen werden, dass die Verbindung geschlossen ist.

Byte	`88 82 (2a 1f fc f2) 29 f7`
Binär	`10001000 10000010 (...) 00101001 11110111`
Detail	`FIN = 1, RSV n. gesetzt`
opcode	`0x8`
Maskierung	`1`
Payload length	`2`
Payload (maskiert)	`)÷`

Abbildung 3: Close-Frame initiiert von Client

Abbildung 3 zeigt einen maskierten Close-Frame, der von einem Client abgesetzt wurde. Der maskierte Payload enthält dabei den Status-Code *1000*. Dieser wurde in der Antwort des Servers ebenfalls zurückgemeldet. Hierbei handelt es sich um die Angabe *Normal Closure*, welche besagt, dass die Verbindung ordnungsgemäß beendet wurde. Der RFC6455 definiert vier Status-Gruppen

Byte	`88 02 03 e8`
Binär	`10001000 00000010 00000011 11101000`
Detail	`FIN = 1, RSV n. gesetzt`
opcode	`0x8`
Maskierung	`0`
Payload length	`2`
Payload (unmaskiert)	`1000`

Abbildung 4: Close-Frame Antwort des Servers

mit folgender Bedeutung [FM11, S. 46]:

0-999 Dieser Bereich wird nicht genutzt.

1000-2999 Diese Status-Codes sind für die Definition innerhalb des RFC6455 sowie für zukünftige Überarbeitungen und Erweiterungen reserviert.

3000-3999 Status-Codes in diesem Bereich sind für Verwendung in Frameworks, Bibliotheken und Anwendungen gedacht und bedürfen einer Registrierung bei der Internet Assigned Numbers Authority (IANA).

4000-4999 In diesem Bereich können eigene Status-Codes definiert werden. Diese benötigen keine Registrierung und müssen den kommunizierenden Anwendungen bekannt sein.

Die Spezifikation des WebSocket-Protokolls definiert bereits einige Status-Codes innerhalb des zweiten Bereichs. Allerdings werden diese nur als Vorschlag angegeben, was inzwischen zu einem gewissen Wildwuchs und damit verbunden zu einigen Problemen führte [GLN15, S. 53]. Bisher wurden nur wenige Status-Codes neben jenen aus der Spezifikation registriert. Die bisher offiziellen Registrierungen finden sich auf den zugehörigen Seiten der IANA.[2] Tabelle 3 (S.13) zeigt einen kurzen Auszug der wichtigsten Status-Codes. Es fällt auf, dass alle Neuregistrierungen durch Alexey Melnikov erfolgten, welcher bereits an der Spezifikation des Protokolls beteiligt war. In einer der betreffenden Anfragen zur Registrierung eines Status-Codes fragt Melnikov auch, ob diese Status-Codes denn überhaupt genutzt werden und ob noch ein Interesse an deren Registrierung besteht [Mel12].

2.2 WebSocket Kommunikationsphasen

Das WebSocket-Protokoll basiert auf dem Transport Control Protocol (kurz: TCP) und durchläuft wie dieses verschiedene Phasen der Kommunikation. Zu Beginn wird ähnlich zu TCP ein Opening-Handshake zum Verbindungsaufbau durchgeführt. Auch am Ende der Datenübertragung wird ein solcher Handshake eingesetzt, um die Verbindung ordnungsgemäßg zu beenden. Die Ports, welche vom WebSocket-Protokoll verwendet werden, sind Port 80 (ungesicherte Verbindungen) und Port 443 (gesicherte Verbindungen). Aktuell wird deren Verwendung jedoch vom

[2] unter https://www.iana.org/assignments/websocket/

Status-Code	Bedeutung	Beschreibung	Spezifiert durch
1000	Normal Closure	Zeigt einen ordnungsgemäßen Verbindungsabbau an.	RFC6455
1002	Protocol Error	Der Endpunkt beendet die Verbindung aufgrund eines Protokollfehlers. Beispielsweise wird der Empfang einer nicht maskierten Nachricht von einem Server durch selbigen mit diesem Status-Code beendet.	RFC6455
1006	Abnormal Closure	Dieser Status darf nicht in einem Close-Frame verwendet werden. Anwendungen können durch diesen signalisieren, dass die Verbindung unerwartet und auf unbekannte Weise beendet wurde.	RFC6455
1010	Mandatory Ext.	Ein WebSocket-Client kann diesen Status-Code verwenden, wenn er die Verbindung aufgrund fehlender Erweiterungen beendet. Ein Server verwendet diesen Status-Code nicht, da er aufgrund mangelnder clientseitiger Erweiterungen den Opening-Handshake abbrechen kann.	RFC6455
1012	Service Restart	Gibt an, dass der Dienst neustartet. Verbundene Clients sollen im Falle einer Wiederherstellung der Verbindung eine zufällige Zeit zwischen 5-30 Sekunden warten [Mel12].	Alexey Melnikov
1013	Try Again Later	Signalisiert, dass der Dienst überlastet ist. Sollte das Ziel über mehrere IP-Adressen verfügen, kann eine andere verwendet werden. Ist nur eine IP-Adresse verfügbar, so soll der Benutzer entscheiden, ob er sich erneut verbinden möchte [Mel12].	Alexey Melnikov

Tabelle 3: Auszug der registrierten Status-Codes

HTTP(S)-Protokoll dominiert und sie müssen daher geteilt werden. Der Verbindungsaufbau wird daher als HTTP-Upgrade Anfrage initiiert. Zukünftige Implementierungen könnten dies jedoch durch einen einfacheren Mechanismus über einen dedizierten Port ablösen, ohne das WebSocket-Protokoll umgestalten zu müssen [FM11, S. 3]. Im Folgenden Abschnitt werden die notwendigen Schritte zum Verbindungsaufbau und -abbau erläutert, sowie ein beispielhafter Ablauf der Übertragung über eine WebSocket-Verbindung gezeigt.

2.2.1 Verbindungsaufbau

Da Verbindungen über die Ports 80 bzw. 443 in den meisten Firewalls zugelassen werden, bedarf es keiner neuen Freigaben bei der Verwendung von WebSockets. Allerdings sind diese Ports oftmals durch einen HTTP-Server belegt. Aus diesem Grund bedarf es beim Aufbau einer WebSocket-Verbindung der Kooperation mit HTTP. Der Client sendet dazu einen HTTP-

Upgrade-Request, der um spezielle Felder erweitert wurde.

```
GET /dako-backend/advancedchat HTTP/1.1
Host: localhost:8080
Connection: Upgrade
Upgrade: websocket
Origin: localhost:8080
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: 1zUsRVYbBDvRKqqZq9cRsg==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
(...)
```

Abbildung 5: Opening-Handshake des Clients

Der Client stellt in Abbildung 5 einen solchen HTTP-Upgrade-Request an den Endpunkt des WebSocket-Servers unter `localhost:8080`. Durch `Connection: Upgrade` wird dem Server mitgeteilt, dass auf ein anderes Protokoll gewechselt werden soll, in diesem Fall `websocket`. Die Versionsnummer entspricht den unterschiedlichen IETF Drafts. Über `Sec-WebSocket-Extensions` werden die Erweiterungen angefragt, die der Client unterstützt. Die übermittelte Erweiterung `permessage-deflate` wird in Kombination mit dem Parameter `client_max_window_bits` angegeben, der die Sliding Window Größe limitiert, mit der der Client die Nachricht komprimiert. So kann der zu reservierende Speicherbedarf bei der serverseitigen Dekompression reduziert werden [Yos15, S. 18]. Der Client generiert den Base64-codierten `Sec-WebSocket-Key`, der anschließend vom Server zur Empfangsbestätigung des Handshakes verwendet wird.

```
HTTP/1.1 101
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: mi8r8dWsVBJS1B3908SLbnd5tHM=
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits=15
(...)
```

Abbildung 6: Opening-Handshake des Servers

Damit der Server bestätigen kann, dass der Handshake eingetroffen ist, wird der übermittelte `Sec-WebSocket-Key` mit dem Globally Unique Identifier (GUID) `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` verbunden und ein SHA-1 Hashwert gebildet. Dieser Wert wird in einer Base64-Kodierung im Feld `Sec-WebSocket-Accept` übermittelt. Der Client kann beim Empfang der Nachricht anhand des gleichen Vorgehens einen SHA-1 Hash erzeugen und mit dem empfangenen vergleichen.

1. Kombinierter Wert `1zUsRVYbBDvRKqqZq9cRsg==258EAFA5-E914-47DA-95CA-C5AB0DC85B11`

2. SHA-1 Hashwert `9A2F2BF1D5AC541252D41DFDD3C48B6E7779B473`

3. Base64 Kodierung `mi8r8dWsVBJS1B3908SLbnd5tHM=`

Des weiteren teilt der Server dem Client mit, dass die Erweiterung `permessage-deflate` unterstützt wird. Die maximale Größe eines Sliding Windows beträgt demnach 2^{15} `Byte` $= 32$ `KB`. Als

HTTP-Status wird der Code `101 Switching Protocols` angegeben. Dieser wird eingesetzt, um zu signalisieren, dass absofort das WebSocket-Protokoll genutzt wird. Weitere Anwendung findet dieser Status-Code beim Wechsel von HTTP-Versionen [Lea+99]. Meldet der Server einen anderen Status, so wird angenommen, dass der WebSocket Handshake nicht vollständig durchgeführt werden konnte und die Semantik von HTTP gilt [FM11, S. 7]. Ebenso wird verfahren, wenn der vom Client ermittelte Hashwert nicht mit dem vom Server übertragenen `Sec-WebSocket-Accept` übereinstimmt.

Optionale Felder können ebenfalls übertragen werden. Hierzu zählt beispielsweise das Header-Feld `Sec-WebSocket-Protocol`, das angibt welches Sub-Protokoll für die Übertragung genutzt werden soll. Hat der Server den Opening Handshake erfolgreich zurückgesendet, so gilt die WebSocket-Verbindung als geöffnet und Daten können gesendet werden.

2.2.2 Datentransfer

Befindet sich die WebSocket-Verbindung nach dem erfolgreichen Opening-Handshake im Status `OPEN`, so können Daten gesendet werden. Wie in Abschnitt 2.1.1 erläutert, definiert das Header-Feld `opcode` eines WebSocket-Frames die übertragene Art von Nutzdaten. Aktuell sind lediglich Text- und Binärdaten offiziell spezifiert.

Byte	81 0e [57 65 62 53 6f 63 6b 65 74 20 32 30 31 38]
Header	FIN = 1, RSV = 0, opcode = 0x1, Payload length = 14
[Payload]	WebSocket 2018

Abbildung 7: Datentransfer von Server zu Client

In Abbildung 7 ist eine Nachricht mit dem Inhalt `WebSocket 2018` von einem Server an einen Client gesendet worden. Als `opcode` wurde `0x1` angegeben, welcher besagt, dass es sich bei der Art des Payloads um eine Textnachricht handelt. Die Größe der Nutzdaten beträgt 14 Byte und die Nachricht konnte ohne Fragmentierung vollständig übertragen werden.

2.2.3 Verbindungsabbau

Der Abbau einer WebSocket-Verbindung erfolgt anhand eines Closing-Handshakes. Er soll den `FIN/ACK` Closing-Handshake von TCP ergänzen, da dieser insbesondere bei der Verwendung von Proxy Servern oder Vermittlungsstellen nicht garantiert zwischen genau zwei Endpunkten abläuft [FM11, S. 9]. Laut RFC6455 sollen so Szenarien vermieden werden, in denen Daten unnötigerweise verloren gehen.

Die Einleitung eines Verbindungsabbaus kann sowohl vom Client, als auch vom Server erfolgen. Dazu wird ein Frame mit dem `opcode`-Wert `0x8` an den Kommunikationspartner versendet. Der versendende Endpunkt wechselt mit dem Absenden des Closing-Frames vom `OPEN` in den `CLOSING` Status. Ebenso gilt dies beim Empfang eines Closing-Frames insofern selbst noch keiner versendet wurde. Danach eintreffende oder zu sendende Nachrichten müssen verworfen werden. Haben

beide Endpunkte sowohl einen Closing-Frame gesendet, als auch empfangen, so gilt die Verbindung als beendet und die darunterliegende TCP-Verbindung kann geschlossen werden. Dies sollte stets vom Server begonnen werden, damit er im `TIME WAIT` Status verbleibt. Der Client kann so die Verbindung durch das Senden eines neuen `SYN` unverzüglich wieder eröffnen [FM11, S. 41].

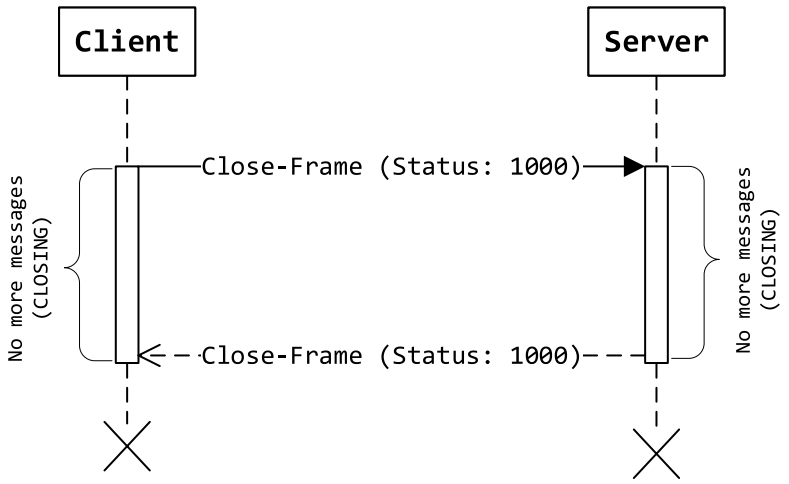

Abbildung 8: Clientseitiger Closing-Handshake

Abbildung 8 zeigt den schematischen Ablauf des Closing-Handshakes. Der Client initiiert diesen, indem er einen Closing-Frame mit dem Statuscode 1000 versendet. Der Server reagiert mit einer entsprechenden Antwort, wobei der Status aus dem ersten Frame übernommen wurde. Nachdem der Closing-Handshake abgeschlossen wurde befinden sich beide Endpunkte im Status `CLOSED` und die zugrundeliegende TCP-Verbindung kann beendet werden.

2.3 Alternative Verfahren

Die Kommunikation mittels HTTP basiert auf einem Request/Response Verfahren. Dabei sendet immer der Client eine Anfrage an einen Server. Dieser antwortet mit einer Statuszeile und entsprechendem Statuscode. Anschließend gilt die Übertragung als beendet und kann durch eine erneute Anfrage des Clients neu gestartet werden. Eine Möglichkeit Daten vom Server an den Client zu senden, ist in der Spezifikation von HTTP nicht vorgesehen. Verändert sich der Zustand einer serverseitigen Ressource, so kann der Client darüber bei einer reinen HTTP-Verbindung nur durch erneutes Anfragen informiert werden. Allerdings benötigen viele Anwendungsfälle diese Funktion. Aus diesem Grund wurde eine Vielzahl von Verfahren entwickelt, die diese Mechanismen abbilden sollen. Im Folgenden werden einige dieser Verfahren kurz vorgestellt und hinsichtlich ihrer Vor- und Nachteile gegenüber WebSocket analysiert.

2.3.1 Polling

Durch die vom W3C standardisierte API XMLHttpRequest (XHR) wurde im Jahr 2012 eine Möglichkeit geschaffen, mittels JavaScript HTTP-Requests zu konstruieren [WHA06]. Ein XHR-Objekt registriert verschiedene Methoden, welche bei einer Änderung des Objekts asynchron aufgerufen werden. Dadurch können Anfragen gesendet werden, ohne den weiteren Programmablauf zu blockieren. Dies bildet die technische Grundlage vieler moderner Web-Frontends, die unter dem Oberbegriff Ajax (*Asynchronous JavaScript and XML*) zusammengefasst werden [GLN15, S. 26].

Beim Polling-Verfahren wird eine Anfrage an den Server in festgelegten Zeitabständen wiederholt. Durch die Verwendung eines XHR-Objekts blockiert die Anwendung zwischen dem Absenden und Empfang der Serverantwort nicht. Liegt keine Änderung vor, so sendet der Server eine leere Antwort zurück.

Dieses Verfahren wird von allen Browsern unterstützt und bedarf keiner besonderen Kenntnisse. Allerdings ist der erzeugte Overhead je nach Einsatzszenario nicht zu unterschätzen. Auch die Belastung des Servers bei der Verarbeitung der eintreffenden Anfragen kann problematisch sein. Treten demnach viele Änderungen in kurzen Zeitabständen auf, so ist von der Verwendung eines einfachen Polling-Verfahrens abzuraten.

2.3.2 Long-Polling

Die regelmäßigen Abfragen des einfachen Pollings führen zu einem enormen Overhead. Um diesen zu reduzieren wird beim Long-Polling nur eine Antwort vom Server zurückgesendet, wenn eine Änderung vorliegt. Der Client reagiert umgehend mit einem neuen Request auf die Antwort des Servers. Die Verbindung bleibt dabei geöffnet, solange keine Änderung auftritt. Läuft die vordefinierte Wartezeit ab, so antwortet der Server auch hier mit einer leeren Nachricht [GLN15, S. 29].

Das Long-Polling reduziert den Overhead gegenüber dem einfachen Polling-Mechanismus deutlich. Auf Grund des sofortigen Wiederaufbaus durch den Client wird eine nahezu andauernde Verbindung simuliert. Dennoch ist auch in diesem Fall der Overhead deutlich größer als bei der Verwendung einer WebSocket-Verbindung. Mit steigender Frequenz von Nachrichten nähert sich dieser an den Overhead des einfachen Pollings.

2.3.3 Comet

Auf Grund diverser Limitierungen für Mehrbenutzeranwendungen, entwickelten sich Programmiermodelle, welche unter der Bezeichnung *Comet* [Rus06] zusammengefasst wurden. Da es sich hierbei nicht um einen spezifizierten Standard handelt, wurde im Rahmen des CometD-Projekts der Dojo Foundation das Bayeux-Protokoll ins Leben gerufen. Ziel des Projekts ist die Reduzierung der Komplexität bei der Implementierung von Comet Anwendungen [Rus+07]. Um dies zu

erreichen wurden Referenzimplementierungen des Bayeux-Protokolls in verschiedenen Programmiersprachen bereitgestellt.

Neben der Verwendung des Long-Polling Mechanismus zur ereignisgesteuerten Nachrichtenübertragung von Server zu Client, sieht das Bayeux-Protokoll auch Streaming-Techniken vor. Dabei wird die HTTP-Verbindung nach der Antwort des Servers auf den Request des Clients nicht geschlossen, sondern anhand des Connection-Headers `keep-alive` weiterhin offen gehalten. Somit können weitere Nachrichten vom Server an den Client gesendet werden, ohne dass eine erneute Anfrage durch den Client gestellt werden muss.

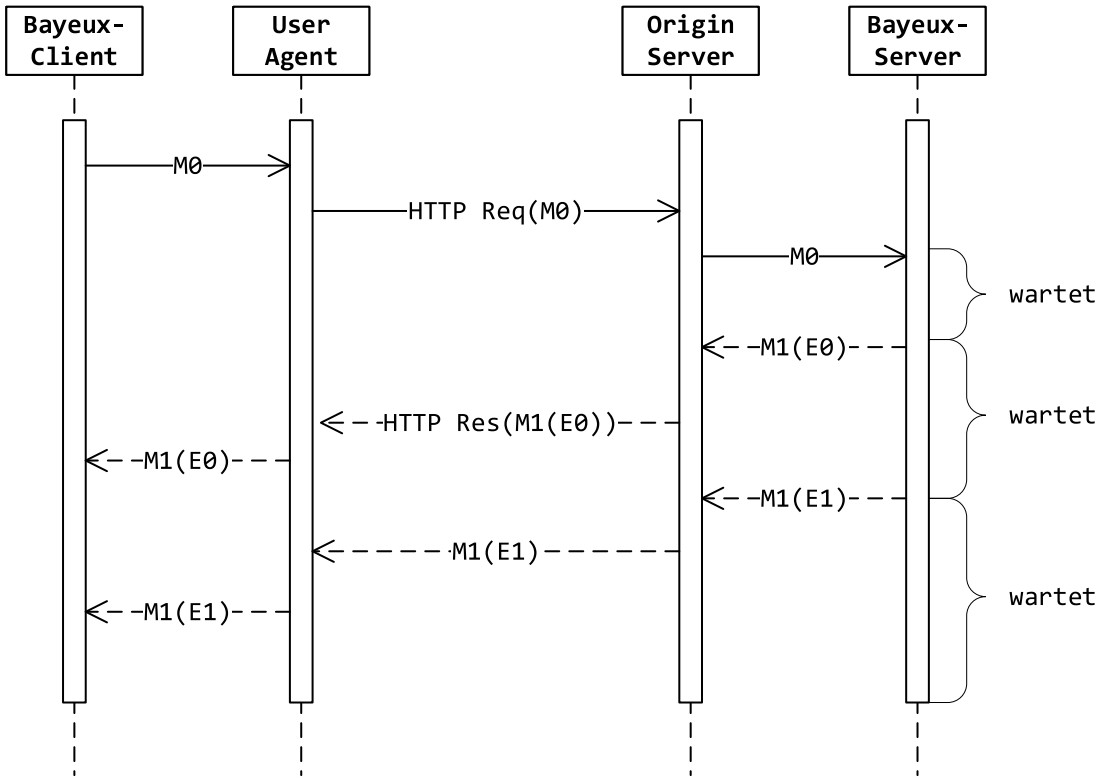

Abbildung 9: HTTP-Streaming im Bayeux-Protokoll

Allerdings muss die Verwendung von HTTP-Streaming von den User Agenten unterstützt werden, da unvollständige HTTP Responses für dieses Verfahren genutzt werden [Rus+07]. Um bidirektional kommunizieren zu können verwendet das Bayeux-Protokoll zwei HTTP-Verbindungen, wohingegen das WebSocket-Protokoll mit einer einzigen Verbindung arbeitet. Der größte Nachteil des Bayeux-Protokolls ist jedoch die fehlende Standardisierung durch eine etablierte Organisation.

2.3.4 Server-Sent Events

Die bislang vorgestellten Verfahren bieten keine standardisierte API, die es Servern ermöglicht Daten über HTTP an eine Website zu senden. In HTML5 wurde dafür vom W3C die sogenannten *Server-Sent Events (SSE)* eingeführt, welche ein `EventSource`-Objekt bereitstellen. Dieses sorgt auf der Clientseite dafür, dass Benachrichtigungen des Servers empfangen werden können. Dazu stellt der Browser drei Event-Handler bereit, die DOM-Events an die Anwendung weiterleiten [GLN15, S. 31]. Bei der Öffnung einer SSE-Verbindung wird der Event-Handler `onopen` aufgerufen, im Falle eines Fehlers der `onerror`-Handler. Kommen Nachrichten des Servers an, so wird der `onmessage`-Handler aktiviert [Hic09a]. Zu jedem Event-Handler werden entsprechende Methoden registriert, welche die Bearbeitung des jeweiligen Events übernehmen.

Server-Sent Events erzeugen keinen Overhead durch wiederholte Anfragen an den Server. Die Kommunikation von Server zu Client wird ermöglicht, ohne mehrere HTTP-Verbindungen zu öffnen. Allerdings handelt es sich hier um einen unidirektionalen Kanal von Server zu Client. Dies ist für Anwendungsfälle ausreichend, in denen lediglich der Server Informationen zu übermitteln hat. Beispiele hierfür wären u.a. das Hardware-Monitoring eines Server oder ein Newsticker. Sollen jedoch Daten in beide Richtungen übertragen werden, so ist die Verwendung von WebSockets die bessere Alternative. Insbesondere Mehrbenutzeranwendungen wie Spiele oder Chats, die auf kurze Latenzen angewiesen sind, profitieren von derem bidirektionalem Kommunikationskanal.

Die Vielfalt der Alternativen lässt darauf schließen, dass ein Bedarf für eine standardisierte Methode besteht, mit der Daten zwischen Client und Server bidirektional übertragen werden können. Dieser Bedarf führte zur Einführung der WebSockets, die als einzige auch für die Übertragung binärer Daten geeignet sind.

3 Die WebSocket-API

Neben der Protokollspezifikation durch das Standardisierungsgremium IETF, wurde auch eine passende API für die Verwendung von WebSockets definiert. Da es sich hierbei um eine Webtechnologie handelt die in Browsern Anwendung findet, obliegt deren Spezifierung dem W3C. Der erste Working Draft vom 23. April 2009 wurde von Ian Hickson eingereicht, einem der führenden Entwickler der Protokollspezifikation [Hic09b]. Darin werden alle benötigten Eigenschaften und Funktionen der WebSocket-Schnittstelle definiert. Die Beschreibung erfolgt anhand der *Web Interface Description Language* (*Web IDL*), einer formalen Darstellung, die an die sprachlichen Konstrukte von ECMAScript bzw. JavaScript angelehnt ist. Dabei handelt es sich um eine Programmiersprache, die 1995 von Netscape Communications entwickelt und durch die European Computer Manufacturers Association (ECMA) unter der Bezeichnung ECMAScript standardisiert wurde [Pey17]. Alle Beispiele beziehen sich auf die aktuelle Version 8, die im Juni 2017 offiziell freigegeben wurde. Im Folgenden wird die geläufigere Bezeichnung JavaScript verwendet.

3.1 Elemente der W3C WebSocket-API

Das Erstellen eines WebSocket-Endpunkts erfolgt laut Spezifikation anhand des Konstruktors. Dieser empfängt die URL, zu welcher der Endpunkt sich verbinden soll und legt diese im read-only Attribut `url` ab. Optional empfängt der Konstruktor einen String oder ein String-Array, welches die unterstützten Sub-Protokolle enthält.

```
var ws = new WebSocket(in DOMString url, in optional DOMString[] protocols);
```

Listing 3-1: WebSocket anlegen und verbinden

Wird die URL nicht im korrekten Schema angegeben, so wird eine `DOMException` mit der Beschreibung `SyntaxError` ausgelöst. Falls Sub-Protokolle angegeben werden, so sendet der Client diese bei der Anfrage im HTTP-Headerfeld `Sec-WebSocket-Protocol`. Wenn Server und Client ein gemeinsames Protokoll vereinbaren konnten, wird es im schreibgeschützten Attribut `protocol` abgelegt. Etwaige Erweiterungen werden nach dem erfolgreichen Verbindungsaufbau im Attribut `extensions` abgelegt.

3.1.1 WebSocket Lebenszyklus

Jede WebSocket-Verbindung unterliegt einem Lebenszyklus. Die Spezifikation unterscheidet hier vier mögliche Zustände, die als konstante Zahlenwerte hinterlegt sind. Je nachdem in welchem Zustand sich der WebSocket gerade befindet, wird der Wert einer dieser Konstanten im Attribut `readyState` abgelegt [WHA10].

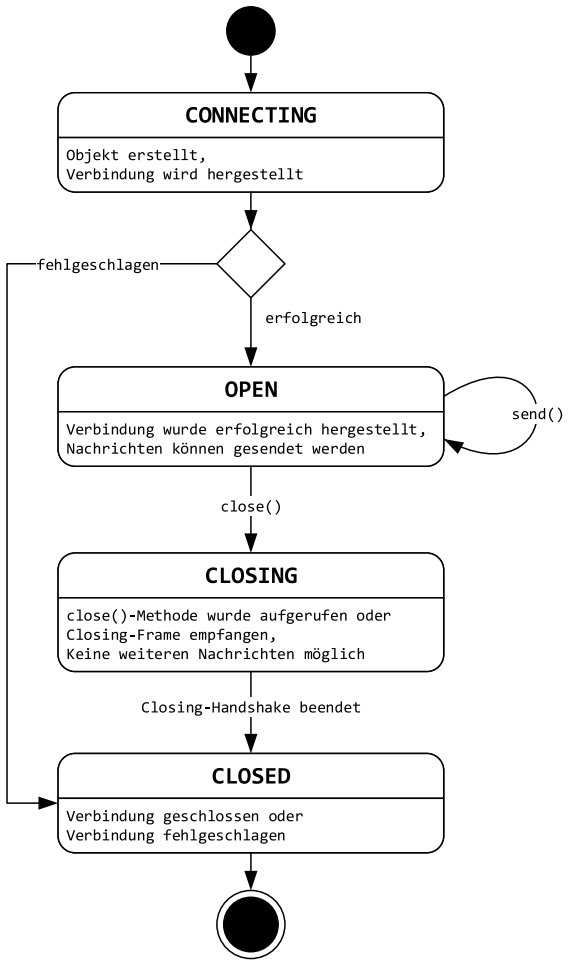

Abbildung 10: UML-Zustandsdiagramm des WebSocket Lifecycle

Die Implementierung in JavaScript spiegelt den Lebenszyklus eines WebSocket-Objekts gemäß der Beschreibung des W3C ab. So wird beim Anlegen des Objekts die Verbindung aufgebaut und lässt anschließend das Senden von Nachrichten zu. Durch den Aufruf der `close()`-Methode wird der Verbindungsaufbau eingeleitet. Das Schließen kann zudem optional mit Angabe eines Status-Codes (Tabelle 3, S.13) und einer textuellen Begründung erfolgen.

```
var ws = new WebSocket("ws://echo.websocket.org"); // 0 = CONNECTING
console.log(ws.readyState); // 1 = OPEN, wenn erfolgreich
ws.send("Message"); // 1
ws.close(1000, "Work done."); // 2 = CLOSING
console.log(ws.readyState); // 3 = CLOSED
```

Listing 3-2: WebSocket Lebenszyklus

3.1.2 Event-Handling

Die Zustände eines WebSockets werden durch Events beeinflusst, die während der Kommunikation mit einem Endpunkt eintreffen. Diese werden von Event-Handlern empfangen und können anschließend verarbeitet werden. Die WebSocket API definiert die vier Event-Handler `onopen`, `onerror`, `onmessage` und `onclose`. Ein erfolgreicher Verbindungsaufbau löst das Event `open` aus, welches vom Event-Handler `onopen` verarbeitet wird. Fehler werden vom `onerror`-Event-Handler signalisiert. Wurde die Verbindung geschlossen oder konnte nicht geöffnet werden, so wird der `onclose`-Event-Handler ausgelöst.

```
var ws = new WebSocket("ws://echo.websocket.org");
// Registrieren des OnOpen-Handlers
ws.onopen = function(){
  console.log("Verbindung hergestellt!");
}
// Registrieren des OnError-Handlers
ws.onerror = function(e){
  console.log("Fehler aufgetreten: " + e.data);
}
// Registrieren des OnMessage-Handlers
ws.onmessage = function(e){
  showMessageInUI(e.data);
}
// Registrieren des OnClose-Handlers
ws.onclose = function(e){
  console.log("CloseEvent erhalten - Verbindung wird abgebaut")
  console.log("Code: " + e.code);
  console.log("Grund: " + e.reason);
}
```

Listing 3-3: WebSocket Event-Handler

3.1.3 Kommunikation

Empfangene Nachrichten können mittels des `onmessage`-Event-Handlers an die Anwendung weitergegeben werden. Entsprechend der Spezifikation des RFC6455 ermöglicht die WebSocket API das Senden von textuellen und binären Daten. Zur Unterscheidung verschiedener Arten binärer Daten stellt sie die Enumeration `BinaryType` bereit. Diese enthält die Werte `blob` und `ArrayBuffer`. Die Übermittlung der Daten erfolgt durch die Methode `send()`. Bei deren Aufruf wird das Attribut `bufferedAmount` um die Anzahl Bytes erhöht, die nicht in das Netzwerk übermittelt werden konnten [WHA10].

```
var ws = new WebSocket("ws://echo.websocket.org");
ws.send("Nachricht"); // als DOMString versenden
ws.send(BlobUtils.fromString("Nachricht")) // als binary blob versenden
```

Listing 3-4: WebSocket Nachrichten senden

3.2 Die Java™ API for WebSocket (JSR-356)

Die WebSocket-APIs beschränken sich nicht nur auf JavaScript. Verschiedene Implementierungen des Standards ermöglichen dem Entwickler, WebSockets in seiner bevorzugten Programmiersprache zu nutzen. Laut des PopularitY of Programming Language (PYPL) Index ist Java der Oracle Corporation die weltweit beliebteste Programmiersprache [Cab17]. Sie wurde 1990 von James Gosling mit seinem Team bei dem Unternehmen Sun Computers entwickelt. Die erste öffentlich zugängliche Version wurde 1995 veröffentlicht [Smy09]. Aktuell ist Java 9 verfügbar.

Java Community Process

Soll eine neue Technologie in den Standardumfang von Java aufgenommen werden, so muss diese den Community Prozess (JCP) durchlaufen [Ora17b]:

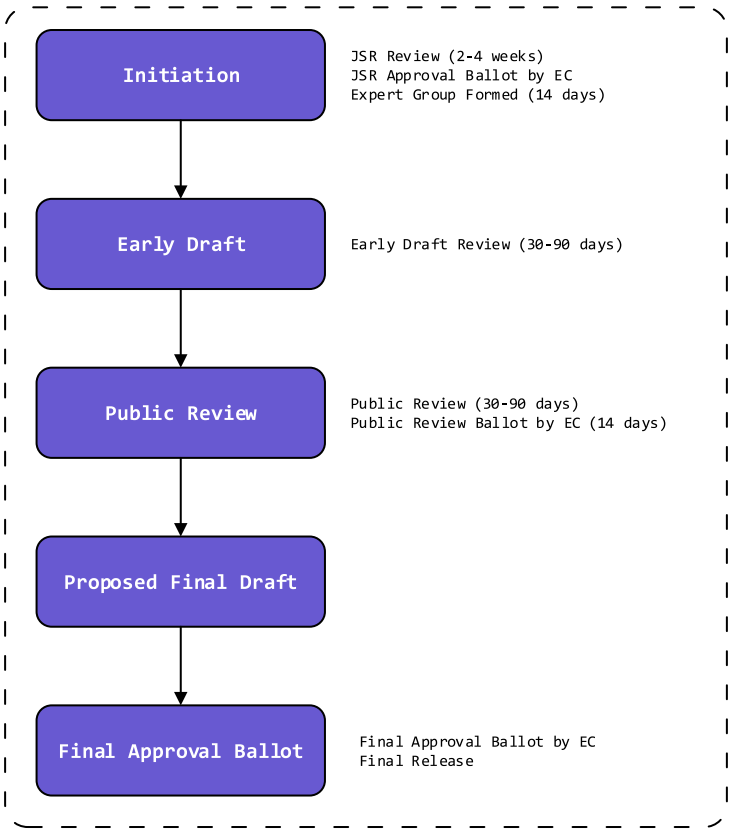

Abbildung 11: JSR Approval Timeline

In Phase 1 können Mitglieder des JCP einen Vorschlag zur Aufnahme einer Erweiterung machen. Diese werden als *Java Specification Request* (JSR) bezeichnet. Wird der JSR vom *Executive Committee* (EC) angenommen, so wird bis Phase 2 eine Expertengruppe gebildet, die einen Early Draft erstellt. Darauf aufbauend entwickelt diese Gruppe einen Public Draft, der in Phase 3 öffentlich eingesehen und kommentiert werden kann. Der Entwurf wird dann entsprechend überarbeitet und eine finale Version eingereicht. Zudem wird eine Referenzimplementierung (RI) und

das Technology Compatibility Kit (TCK) erstellt, welches Tests und Tools bereitstellt um Implementierungen auf Konformität mit dem JSR zu prüfen. Abschließend stimmt das EC noch über die Aufnahme in die Standardbibliothek ab. Befindet sich der JSR dann im produktiven Betrieb, können in der Maintenance Phase Updates und Änderungen des JSR beantragt und eingepflegt werden [Ora17b].

Der JSR 356 trägt die Bezeichnung Java™API for WebSocket und regelt die Aufnahme der WebSocket-Technologie in die Standardbibliothek der Java Enterprise Edition (JEE). Er befindet sich aktuell in der 4. Phase und ist seit dem 22. Mai 2013 offizieller Bestandteil seit JEE 7. Die folgenden Beispiele verwenden die achte Version von Java.

3.2.1 Grundlagen

Alle Klassen und Interfaces zur Impementierung von WebSocket-Clients und Servern wurden für Java in dem Paket `javax.websocket` zusammengefasst. Für einen WebSocket-Client, der weder im Browser noch in einem anderen Framework zum Einsatz kommt, kann in Java die Open-Source Referenzimplementierung *Tyrus* von Oracle verwendet werden. Nachfolgende Beispiele verwenden das aktuellste Release 1.12. Die Bereitstellungen erfolgen über Maven:

```
<dependency>
      <groupId>javax.websocket</groupId>
      <artifactId>javax.websocket-api</artifactId>
      <version>1.1</version>
      <scope>provided</scope>
</dependency>
<dependency>
      <groupId>org.glassfish.tyrus.bundles</groupId>
      <artifactId>tyrus-standalone-client</artifactId>
      <version>1.12</version>
</dependency>
```

Für die Implementierung eines WebSocket-Serverendpunkts wird zudem ein Webserver benötigt, der die WebSocket-API implementiert. Für Java gibt es hier eine Vielzahl von Alternativen zur Auswahl:

- Apache Tomcat ab Version 7.0.56 [The14]
- GlassFish Server ab Version 4.1 [Ora14]
- Grizzly ab Version 2 [Ora17a]
- Eclipse Jetty ab Version 7 [The16]
- Netty ab Version 4.0 [Lee11]

Ansätze der Programmierung

Die Java™API for WebSocket kann auf zwei unterschiedliche Weisen verwendet werden. Der programmatische Ansatz sieht die Vererbung der `Endpoint` Klasse der API vor. Dabei müssen u.a. die Methoden der Oberklasse überschrieben und Funktionen auf Events registriert werden. [Cow14, S. 9]:

```
public class MyWebSocketClient extends Endpoint {
  @Override
  public void onOpen(Session session) {
    session.addMessageHandler(new MessageHandler.Whole<String>() {
      @Override
      public void onMessage(String message) {
        // Verarbeitung der Nachricht
      }
    });
  }
}
```

Listing 3-5: Java: Programmatischer Ansatz

Der Einsatz von Annotationen transformiert einfache Java-Objekte zu WebSocket Endpunkten. Handelt es sich um eine Annotation auf Klassenebene, so können diverse Konfigurationsparameter angegeben werden. Alle Informationen sind somit an einer Stelle gesammelt und auf eine Konfigurationsklasse kann in vielen Fällen verzichtet werden.

```
@ServerEndpoint( value = "/endpoint", subprotocols = {"chat"})
public class MyWebSocketServer { ... }
```

Listing 3-6: Java: Annotationen

Die nachfolgenden Code-Beispiele sind immer annotations-basiert, aufgrund besserer Lesbarkeit, verkürzter Schreibweise und der einfachen Verständlichkeit.

Erzeugung eines WebSockets

Clientseitige Endpunkte werden mit der Annotation `@ClientEndpoint` versehen. Die Konstruktion eines WebSockets erfolgt hier durch die Erzeugung eines Container-Objekts, welches anschließend zum Verbindungsaufbau verwendet wird.

```
@ClientEndpoint
public class MyClient {
  public MyClient(URI hostPath){
    // Container-Objekt erstellen
    WebSocketContainer wsc = ContainerProvider.getWebSocketContainer();
    // Verbindung fuer Endpunkt MyClient herstellen
    wsc.connectToServer(this, hostPath);
  } }
```

Listing 3-7: Java Client: Verbindungsaufbau

Die Erstellung des Objekts ist nur dann erfolgreich, wenn die WebSocket-Verbindung aufgebaut werden konnte. Ein WebSocket-Server wird mit der Annotation `@ServerEndpoint` gekennzeichnet. Sowohl bei Server- als auch bei Clientendpunkten können verschiedene Konfigurationsparameter verwendet werden (siehe Listing 3-6). Der Parameter `value` gibt dabei den relativen Pfad an, unter dem der WebSocket-Endpunkt erreichbar ist. Wird die Anwendung auf einem lokalen WebServer ausgeführt und ihre Bezeichnung lautet `example`, so ist die Anwendung erreichbar unter `ws://localhost[:port]/example/endpoint`. Unterstützte Subprotokolle können optional mittels des Parameters `subprotocols` angegeben werden. Falls die Anwendung eigene Java-Objekte oder komplexere Datentypen übertragen soll, so müssen Encoder- bzw. Decoder-Klassen angegeben werden. Diese werden im Abschnitt 3.2.3 näher erläutert.

Event-Handling

Durch die Verwendung der Annotationen erfolgt in Java keine manuelle Deklaration der Handler. Eine Methode, die ein bestimmtes Event bearbeitet, wird durch Angabe der jeweiligen Annotation zu einem EventHandler transformiert.

```
@OnOpen
public void init(Session session, EndpointConfig config){
  // Session und Konfiguration in Variable speichern
}

@OnError
public void error(Throwable t) {
  System.out.println("Fehler aufgetreten: " + t.getMessage());
}

@OnClose
public void close(CloseReason c) {
  System.out.println("Die Verbindung wurde geschlossen.");
  System.out.println("Code: " + c.getCloseCode());
  System.out.println("Grund: " + c.getReasonPhrase());
}

@OnMessage
public void incoming(String msg){
  System.out.println("Neue Nachricht empfangen: " + msg);
}
```

Listing 3-8: Java: EventHandler

Die Java™ API for WebSocket sieht dafür die Annotationen `@OnOpen`, `@OnError`, `@OnMessage` und `@OnClose` vor. Jede auf diese Weise deklarierte Methode ermöglicht die Angabe einiger Parameter. Die aktuell verwendete `Session` kann dabei immer empfangen werden.

3.2.2 Kommunikation

Der RFC6455 definiert Ping- und Pong-Frames, durch die beispielsweise geprüft werden kann, ob ein Endpunkt noch erreichbar ist oder um eine inaktive Verbindung aufrechtzuerhalten. Allerdings spezifiert die W3C WebSocket API keine Möglichkeit, diese Art von Control-Frames zu versenden. Die Java API hingegen stellt dafür die Methoden `sendPing()` und `sendPong()` zur Verfügung, die ein `ByteBuffer`-Objekt als Parameter empfangen, über welches Nutzdaten mit bis zu 125 Byte angehängt werden können. Diese Limitierung wird aufgrund der im RFC6455 definierten Größe von Control-Frames durch eine `IllegalArgumentException` sichergestellt.

```
public void sendPing(ByteBuffer applicationData) throws IOException,
    IllegalArgumentException

public void sendPong(ByteBuffer applicationData) throws IOException,
    IllegalArgumentException

@OnMessage
public void catchPong(PongMessage pong, Session session){ ... }
```

Listing 3-9: Java: Ping/Pong Nachrichten

Die Antwort auf eine eingehende Ping-Nachricht muss nicht durch den Entwickler erfolgen. Die Implementierungen der Java WebSocket-API stellen sicher, dass eine korrespondierende Pong-Nachricht so bald wie möglich zurückgesendet wird. In Listing 3-9 ist die Methode `catchPong` definiert, die als Parameter ein `PongMessage`-Objekt empfängt. Durch sie ist es möglich, die Antwort einer zuvor gesendeten Ping-Nachricht weiter zu verarbeiten. Beispielsweise können so Messungen der *Round Trip Time* (RTT) durchgeführt werden. Allerdings ist deren Aussagekraft nur bedingt gegeben, da diese bei größeren Nachrichten stark abweichen können [Cow14, S. 116]. Das Senden einer Pong-Nachricht ohne vorangegangenen Ping kann genutzt werden, um einen unidirektionalen *Heartbeat* abzubilden. Ein Endpunkt antwortet spezifikationsgemäß nicht auf eine eintreffende Pong-Nachricht, es sei denn, eine entsprechende Methode wurde dafür bereitgestellt.

Das Senden und Empfangen sowohl textueller als auch binärer Daten ist ebenfalls in der Java WebSocket API definiert. Die grundlegendsten Datentypen, die dabei eingesetzt werden können sind `String`, `ByteBuffer` und `byte[]` [Cow14, S. 30]. Darüber hinaus ist es möglich, komplexere Datenstrukturen oder eigene Objekte zu versenden. Dies setzt jedoch entsprechende Decoder- und Encoder-Klassen voraus. Liegen Daten zum Zeitpunkt des Versendens noch nicht vollständig vor oder können aufgrund ihrer Größe nicht in einem einzigen Frame übertragen werden, so kann auf Methoden zurückgegriffen werden, die das partielle Übertragen von Nachrichten ermöglichen. Aufgrund der protokolleigenen Fragmentierung können diese Teilnachrichten von den tatsächlich übertragenen Fragmenten abweichen. Wird der letzte Teil einer Nachricht übertragen, so muss das `isLast` Flag der jeweiligen `send`-Methode gesetzt werden. Das Versenden erfolgt immer über die `RemoteEndpoint` Schnittstelle, die man durch ein `Session` Objekt erhält.

```
public void sendMessages(Session session){

  RemoteEndpoint.Basic rEndpoint = session.getBasicRemote();

  // Textnachricht als String
  rEndpoint.sendText("Message");

  // Binaere Daten senden
  rEndpoint.sendBinary(ByteBuffer.wrap("Message".getBytes()));

  // Partielles Senden einer Textnachricht
  String part1 = "Mess";
  String part2 = "age";
  boolean isDone = false;
  rEndpoint.sendText(part1, isDone);
  rEndpoint.sendText(part2, !isDone);
}
```

Listing 3-10: Java: Synchrone Nachrichtenübertragung

Die bisher vorgestellten Methoden zum Nachrichtenversand blockieren solange, bis der gesamte Payload übertragen wurde. Soll die Applikation hingegen weitere Aufgaben während des Sendens erledigen können, so muss auf die Funktionen der Schnittstelle `RemoteEndpoint.Async` zurückgegriffen werden. Beispiele für den Einsatz asynchroner Verarbeitung finden sich häufig. So soll eine grafische Oberfläche beim Versand einer sehr großen Nachricht weiterhin reagieren können. Um Nachrichten asynchron zu übertragen, stellt die API zwei Varianten zur Verfügung.

```
// Liefert Future-Objekt zurueck
public Future<Void> sendText(String text);

// Benoetigt SendHandler
public void sendText(String text, SendHandler handler);
```

Listing 3-11: Java: Future und Handler

Der erste Weg besteht aus dem Senden einer WebSocket-Nachricht *by future* [Cow14, S. 119]. Der Sinn und Zweck eines `Future`-Objekts ist die Nachverfolgung des Versendungsstatus durch den Aufruf der zugehörigen Methoden. So kann mittels `isDone()` geprüft werden, ob die Nachricht versand wurde. Die Methode `cancel()` bricht eine ausstehende Übertragung ab. Im Unterschied zur synchronen Nachrichtenübertragung werden etwaige Ausnahmen oder Fehler nicht automatisch zurückgemeldet. Diese können nur durch `Future.get()` ermittelt werden. Diese Methode blockiert die weitere Ausführung, bis die Nachricht übermittelt wurde oder ein Fehler aufgetreten ist.

Die zweite Möglichkeit Nachrichten asnychron zu übermitteln erfordert die Bereitstellung eines *callback* Objekts. Dabei handelt es sich um eine Implementierung der `SendHandler`-Schnittstelle,

die beim Eintreffen eines Ereignisses ausgelöst wird. Man spricht vom Senden einer Nachricht *with handler*.

```
public class MyHandler implements SendHandler {
  @Override
  public void onResult(SendResult arg0) {
    // Verarbeiten des Ergebnisses
  }
}
```

Listing 3-12: Java: Callback Interface SendHandler

Beide Varianten der asynchronen Nachrichtenübermittlung können sowohl textuelle, als auch binäre Daten übertragen. Für das Senden komplexer Strukturen oder eigener Objekte wird auch hier ein Encoder benötigt. Ping- bzw. Pong-Nachrichten können nicht asynchron übertragen werden. Die Notwendigkeit asynchroner Übertragung ist in diesem Fall auch nicht gegeben, da diese Control-Frames maximal 125 Byte Nutzdaten enthalten können und dementsprechend selbst bei geringen Bandbreiten unverzüglich übermittelt werden [Cow14, S. 120].

3.2.3 En- und Decoding

Die W3C WebSocket-API definiert lediglich textuelle und binäre Datentypen zur Übermittlung von Nutzdaten und entspricht damit den Anforderungen, die im RFC6455 für das WebSocket-Protokoll festgelegt wurden. Die Java WebSocket-API stellt hierfür standardisierte Methoden bereit, die das Senden eines `String`-Objekts oder binärer Datentypen wie `ByteBuffer` bzw. `byte[]` ermöglichen. In vielen Fällen ist es jedoch von Vorteil, komplexere Datentypen oder eigene Objekte übertragen zu können. Dafür wurde die `send`-Methode der `RemoteEndpoint`-Schnittstelle überladen, sodass sie einen Parameter vom Typ `Object` empfängt. Mit Hilfe dieser Methode lassen sich demnach Objekte eines beliebigen Datentyps versenden. Da das WebSocket-Protokoll aber lediglich textuelle oder binäre Daten kennt, müssen alle komplexeren Objekte in eine dieser beiden Formen gebracht werden. Zu diesem Zweck wurden die Schnittstellen `Encoder` und `Decoder` im Paket `javax.websocket` bereitgestellt. Über untergeordnete Schnittstellen kann zwischen der Umwandlung in textuelle oder binäre Daten gewählt werden.

```
  public interface Encoder {
    // Aufruf bei Verbindungsaufbau
    void init(EndpointConfig config);
    // Aufruf bei Verbindungsabbau
    void destroy();
    // Sub-Interfaces mit jeweiliger encode-Methode
    interface Text<T> extends Encoder { String encode(T object) throws
        EncodeException; }
    interface Binary<T> extends Encoder { ByteBuffer encode(T object) throws
        EncodeException; }
  }
```

Listing 3-13: Java: Encoder Interface

Die textuelle Enkodierung eines Objekts mit der Klassenbezeichnung `MyObject` erfolgt demnach über eine Klasse, die das `Encoder.Text<MyObject>` Interface implementiert. Sowohl die Enkodierung, als auch die Dekodierung erfolgen automatisch und müssen nicht vom Entwickler ausgelöst werden. Die Enkodierung von `MyObject` erfolgt beim Aufruf der `send`-Methode.

```
public interface Decoder {

  // Aufruf bei Verbindungsaufbau
  void init(EndpointConfig config);

  // Aufruf bei Verbindungsabbau
  void destroy();

  // Sub-Interfaces mit jeweiliger decode- und willDecode-Methode
  interface Binary<T> extends Decoder {
    T decode(ByteBuffer bytes) throws DecodeException;
    boolean willDecode(ByteBuffer bytes);
  }

  interface Text<T> extends Decoder {
    T decode(String s) throws DecodeException;
    boolean willDecode(String s);
  }
}
```

Listing 3-14: Java: Decoder Interface

Ein Endpunkt, der nur über eine `@OnMessage`-Methode verfügt, die als Parameter ein Objekt vom Typ `MyObject` empfängt, muss beim Erhalt einer textuellen Nachricht diese dekodieren. Dies erfolgt über eine Klasse, die das `Decoder.Text<MyObject>`-Interface implementiert hat. Dazu ist es zunächst notwendig, zu überprüfen ob die Dekodierung möglich ist. Dies wird über die Methode `willDecode` sichergestellt. Anschließend wird das erwartete Objekt aus dem empfangenen Text dekodiert.

Damit ein WebSocket-Endpunkt Nachrichten enkodieren oder dekodieren kann, müssen diese Klassen bekannt gemacht werden. Dies erfolgt durch deren Angabe als Parameter in der jeweiligen Klassenannotation. Sollen verschiedene Objekte versendet oder empfangen werden, so müssen entsprechende Encoder- und Decoderklassen bereitgestellt werden. Die Auswahl der passenden Klasse erfolgt anhand ihrer Reihenfolge in der Klassenannotation.

```
@ClientEndpoint(encoders = { MyObjectEncoder.class },
                decoders = { MyObjectDecoder.class })
public class MyWebSocketClient { ... }
```

Listing 3-15: Java: Angabe der Encoder- und Decoder-Klassen

Jede WebSocket-Verbindung erhält eine eigene Instanz der Encoder- und Decoder-Klassen. Dadurch wird sichergestellt, dass niemals mehr als ein Thread in der `decode()` oder `encode()` Methode aktiv ist [Cow14, S. 82].

3.2.4 Konfiguration und `Session`

Eine der zentralen Komponenten der Java WebSocket-API ist die `EndpointConfig` Schnittstelle. Durch sie können einem Endpunkt diverse Meta-Daten übermittelt werden, die den gemeinsamen Zustand aller verbundenen Endpunkte repräsentieren. Je nach Art des Endpunktes wird weiterhin zwischen `ClientEndpointConfig` und `ServerEndpointConfig` unterschieden. Die Parameter sind bei der Verwendung von Annotationen in der jeweiligen Klassenannotation anzugeben. Sie werden zum Zeitpunkt der Bereitstellung dazu verwendet, um eine entsprechende Instanz der zugehörigen Endpunktkonfiguration zu erstellen. Diese Konfiguration kann als optionaler Parameter von den Event-Handlern empfangen werden.

Die Methoden der übergeordneten `EndpointConfig` Schnittstelle sind sowohl für Server-, als auch für Clientendpunkte verfügbar. Durch sie können die konfigurierten Encoder- und Decoder-Klassen sowie ein `Map` Objekt abgerufen werden. Die `Map` kann vom Entwickler genutzt werden, um applikationsspezifische Objekte und Informationen abzulegen.

```
public interface EndpointConfig{
  // Liste aller Encoder-Klassen
  List<Class<? extends Encoder>> getEncoders()
  // Liste aller Decoder-Klassen
  List<Class<? extends Decoder>> getDecoders()
  // Editierbare Map
  Map<String,Object> getUserProperties()
}
```

Listing 3-16: Java: EndpointConfig

Die Konfigurationsschnittstelle des Clients ermöglicht den Zugriff auf clientspezifische Eigenschaften wie die bevorzugten Subprotokolle, Erweiterungen und den Konfigurator. Der Konfigurator ist eine Klasse, die Modifikationen an der HTTP-Anfrage sowie eine manuelle Auswertung der zugehörigen Antwort des Servers ermöglicht.

Subprotokolle und Erweiterungen Die Spezifikation des WebSocket-Protokolls ermöglicht die Bereitstellung von Subprotokollen und Erweiterungen. Die verwendeten HTTP-Felder im Opening-Handshake lauten `Sec-WebSocket-Protocol` und `Sec-WebSocket-Extensions`. Subprotokolle werden dabei als einfache `Strings` angegeben und durch Kommata getrennt. Bei der Vergabe eines eigenen Namens für ein Subprotokoll sollte beachtet werden, dass es nicht zu einer Überschneidung mit den bei der IANA registrierten Bezeichnungen kommt. Werden WebSocket-Frames um zusätzliche Informationen erweitert, um beispielsweise den Inhalt der Nutzdaten näher zu beschreiben, so spricht man von `Extensions` [Cow14, S. 102]. Verwendet eine Applikation Subprotokolle oder Erweiterungen, so werden diese ebenfalls in der Klassenannotation hinterlegt.

Für das Hinzufügen oder Ändern der HTTP-Header muss die verwendete Konfiguratorklasse von der Klasse `ClientEndpointConfig.Configurator` abgeleitet werden. Ein häufiger Anwendungsfall ist die Erweiterung der HTTP-Anfrage um applikations- oder frameworkspezifische Informationen für zusätzliche Funktionen [Cow14, S. 104]. In Listing 3-17 ist ein solcher Fall beispielhaft dargestellt.

```
public class MyClientConfig extends ClientEndpointConfig.Configurator {

  @Override
  public void afterResponse(HandshakeResponse hr) {
    // Pruefen, ob Feature aktiviert werden konnte
    boolean activated = hr.getHeaders().get("My-Feature-Field").get(0)
                        .equals("activated");

    if(activated){
      // MyFeature nutzen
    } else {
      // Auf Fallback-Mechanismus zurueckgreifen
    }
    super.afterResponse(hr);
  }

  @Override
  public void beforeRequest(Map<String, List<String>> headers) {
    // HTTP-Headerfeld fuer "MyFeature" erzeugen
    List<String> featureParam = new ArrayList<String>();
    featureParam.add("enable");

    // HTTP-Header erweitern
    headers.put("My-Feature-Field", featureParam);

    super.beforeRequest(headers);
  }
}
```

Listing 3-17: Java: ClientEndpointConfig Konfigurationsklasse

Die Konfiguration des Server durch die `ServerEndpointConfig` Schnittstelle bietet ebenfalls den Zugriff auf einen Konfigurator. Zudem ist es möglich die unterstützten Subprotokolle und Erweiterungen des Servers sowie die Klasse des Endpunkts abzurufen. Die Ermittlung des relativen Pfads des Serverendpunkts wird in Abschnitt 3.2.5 detailliert behandelt.

Der Konfigurator der `ServerEndpointConfig` stellt zusätzlich diverse Methoden bereit, über die der Verbindungsaufbau gezielt gesteuert werden kann. So kann ein passendes Subprotokoll oder eine Erweiterung gewählt, die Erzeugung einer neuen Instanz manuell durchgeführt und in den Handshake selbst eingegriffen werden. Die Methode `checkOrigin()` prüft, ob der für Browserclients verbindliche HTTP-Origin-Header auf den selben Webserver verweist, auf dem der Serverendpunkt ausgeführt wird [Cow14, S. 107]. Eine Anpassung ist meist nicht notwendig. Listing

3-18 zeigt den Konfigurator des Serverendpunkts, welcher den modifizierten Opening-Handshake des Clients aus Listing 3-17 bearbeitet. Dazu wird in der Methode `modifyHandshake()` die Anfrage des Clients auf das entsprechende Feld geprüft und die Antwort des Servers vor dem Versenden angepasst.

```
public class MyServerConfig extends ServerEndpointConfig.Configurator {
  @Override
  public boolean checkOrigin(String originHeaderValue) {
    // Hier kann die Pruefung des Origin-Headers durchgefuehrt werden
  }

  @Override
  public <T> T getEndpointInstance(Class<T> endpointClass) throws
      InstantiationException {
    // Hier kann die Erzeugung einer neuen Instanz angepasst werden
  }

  @Override
  public List<Extension> getNegotiatedExtensions(List<Extension> installed,
      List<Extension> requested) {
    // Hier kann der Standardablauf der Wahl der Erweiterung veraendert werden
  }

  @Override
  public String getNegotiatedSubprotocol(List<String> supported, List<String>
      requested) {
    // Hier kann der Standardablauf der Protokollwahl veraendert werden
  }

  @Override
  public void modifyHandshake(ServerEndpointConfig sec, HandshakeRequest
      request, HandshakeResponse response) {
    // Anfrage auf zusaetzliches Header-Feld pruefen
    boolean activateMyFeature = request.getHeaders().get("My-Feature-Field").
        get(0).equals("enable");

    if(activateMyFeature) {
      // HTTP-Headerfeld fuer "MyFeature" erzeugen
      List<String> featureParam = new ArrayList<String>();
      featureParam.add("activated");

      // HTTP-Antwort-Header erweitern
      response.getHeaders().put("My-Feature-Field", featureParam);
    }
  }
}
```

Listing 3-18: Java: ServerEndpointConfig Konfigurationsklasse

3.2.5 Path-Mapping

4 Fachliche Anforderungen

4.1 Analyse des DaKo-Frameworks

4.2 Anforderungen an WebSocket-Chat

5 Technische Anforderungen

5.1 Entwurf des WebSocket-Chats

5.2 Implementierung des WebSocket-Chats

6 Evaluation

6.1 Test-Umgebung

6.2 Leistungsanalyse WebSocket-Chat

7 Zusammenfassung und Ausblick

7.1 Einsatzgebiete

Literatur

[Abt15] Dietmar Abts. *Masterkurs Client/Server-Programmierung mit Java: Anwendungen entwickeln mit Standard-Technologien*. 4. Aufl. Lehrbuch. OCLC: 913050062. Wiesbaden: Springer Vieweg, 2015. 341 S. ISBN: 978-3-658-09920-6 978-3-658-09921-3.

[Cab17] Pierre Cabonnelle. *PYPL PopularitY of Programming Language Index*. 1. Dez. 2017. URL: http://pypl.github.io/PYPL.html (besucht am 31. 12. 2017).

[Cow14] Danny Coward. *Java WebSocket Programming*. New York: McGraw-Hill, 2014. 230 S. ISBN: 978-0-07-182719-5.

[Fet10] Ian Fette. *The WebSocket Protocol*. 31. Aug. 2010. URL: https://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-01 (besucht am 19. 12. 2017).

[FM11] Ian Fette und Alexey Melnikov. *The WebSocket Protocol*. 1. Dez. 2011. URL: https://tools.ietf.org/html/rfc6455 (besucht am 19. 12. 2017).

[GLN15] Peter Leo Gorski, Luigi Lo Iacono und Hoai Viet Nguyen. *WebSockets: moderne HTML5-Echtzeitanwendungen entwickeln*. OCLC: 901007083. München: Hanser, 2015. 269 S. ISBN: 978-3-446-44438-6.

[Hic09a] Ian Hickson. *Server-Sent Events*. 23. Apr. 2009. URL: https://www.w3.org/TR/2009/WD-eventsource-20090423/ (besucht am 28. 12. 2017).

[Hic09b] Ian Hickson. *The Web Sockets API*. 23. Apr. 2009. URL: https://www.w3.org/TR/2009/WD-websockets-20090423/ (besucht am 30. 12. 2017).

[Hic10] Ian Hickson. *The WebSocket Protocol*. 6. Mai 2010. URL: https://tools.ietf.org/search/draft-hixie-thewebsocketprotocol-76 (besucht am 19. 12. 2017).

[Hua+11] Lin-Shung Huang u. a. „Talking to Yourself for Fun and Profit“. In: *Proceedings of W2SP* (1. Jan. 2011), S. 1–11.

[Int11] Internet Assigned Numbers Authority. *WebSocket Protocol Registries*. 21. Okt. 2011. URL: https://www.iana.org/assignments/websocket/websocket.xhtml#opcode (besucht am 25. 12. 2017).

[Lea+99] Paul J. Leach u. a. *Hypertext Transfer Protocol – HTTP/1.1*. 1. Juni 1999. URL: https://tools.ietf.org/html/rfc2616 (besucht am 20. 12. 2017).

[Lee11] Trustin Lee. *Netty.News: WebSockets in Netty*. 17. Nov. 2011. URL: https://netty.io/news/2011/11/17/websockets.html (besucht am 04. 01. 2018).

[Mel12] Alexey Melnikov. *Additional WebSocket Close Error Codes*. 16. Mai 2012. URL: http://www.ietf.org/mail-archive/web/hybi/current/msg09670.html (besucht am 25. 12. 2017).

[Ora14] Oracle Corporation. *GlassFish Server Open Source Edition Release Notes 4.1*. 1. Sep. 2014. URL: https://javaee.github.io/glassfish/doc/4.0/release-notes.pdf (besucht am 04. 01. 2018).

[Ora17a] Oracle Corporation. *Project Grizzly - WebSockets Overview*. 6. Sep. 2017. URL: https://javaee.github.io/grizzly/websockets.html#/Overview_of_Grizzlys_WebSocket_Implementation (besucht am 04. 01. 2018).

[Ora17b] Oracle Corporation. *The Java Community Process(SM) Program - Introduction - Timeline View*. 1. Jan. 2017. URL: https://jcp.org/en/introduction/timeline (besucht am 31. 12. 2017).

[Pey17] Sebastian Peyrott. *A Brief History of JavaScript*. 16. Jan. 2017. URL: https://auth0.com/blog/a-brief-history-of-javascript/ (besucht am 31. 12. 2017).

[Rus+07] Alex Russell u. a. *The CometD Reference Book – 3.1.3*. 1. Jan. 2007. URL: https://docs.cometd.org/current/reference/#_bayeux (besucht am 28. 12. 2017).

[Rus06] Alex Russell. *Comet: Low Latency Data for the Browser – Infrequently Noted*. 3. März 2006. URL: https://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/ (besucht am 28. 12. 2017).

[She17] Eric Shepherd. *Writing WebSocket Servers*. 16. Aug. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers (besucht am 22. 12. 2017).

[Smy09] Tamara Smyth. *Brief History of Java*. 1. März 2009. URL: https://www.cs.sfu.ca/~tamaras/javaIntro/Brief_History_Java.html (besucht am 31. 12. 2017).

[The14] The Apache Foundation. *Apache Tomcat 7 (7.0.56) - Changelog*. 6. Okt. 2014. URL: http://tomcat.apache.org/tomcat-7.0-doc/changelog.html (besucht am 04. 01. 2018).

[The16] The Eclipse Foundation. *Jetty - Servlet Engine and Http Server*. 1. Jan. 2016. URL: http://www.eclipse.org/jetty/about.html (besucht am 04. 01. 2018).

[WHA06] WHATWG. *XMLHttpRequest Standard*. 1. Jan. 2006. URL: https://xhr.spec.whatwg.org/ (besucht am 27. 12. 2017).

[WHA10] WHATWG. *HTML Standard WebSockets*. 2010. URL: https://html.spec.whatwg.org/multipage/web-sockets.html (besucht am 19. 12. 2017).

[Yos15] Takeshi Yoshino. *Compression Extensions for WebSocket*. 24. Aug. 2015. URL: https://tools.ietf.org/html/draft-ietf-hybi-permessage-compression-28#section-4 (besucht am 26. 12. 2017).

Abbildungsverzeichnis

1 Vereinfachte Darstellung eines WebSocket Frames 6
2 Ping- und Pong-Frames zur Verbindungskontrolle 11
3 Close-Frame initiiert von Client 11
4 Close-Frame Antwort des Servers 12
5 Opening-Handshake des Clients 14
6 Opening-Handshake des Servers 14
7 Datentransfer von Server zu Client 15
8 Clientseitiger Closing-Handshake 16
9 HTTP-Streaming im Bayeux-Protokoll 18
10 UML-Zustandsdiagramm des WebSocket Lifecycle 21
11 JSR Approval Timeline 23

Tabellenverzeichnis

1 Maskierung durch bitweise XOR-Verknüpfung 10
2 Demaskierung durch bitweise XOR-Verknüpfung 10
3 Auszug der registrierten Status-Codes 13

Listings

3-1 WebSocket anlegen und verbinden . . . 20
3-2 WebSocket Lebenszyklus . . . 21
3-3 WebSocket Event-Handler . . . 22
3-4 WebSocket Nachrichten senden . . . 22
3-5 Java: Programmatischer Ansatz . . . 25
3-6 Java: Annotationen . . . 25
3-7 Java Client: Verbindungsaufbau . . . 25
3-8 Java: EventHandler . . . 26
3-9 Java: Ping/Pong Nachrichten . . . 27
3-10 Java: Synchrone Nachrichtenübertragung . . . 28
3-11 Java: Future und Handler . . . 28
3-12 Java: Callback Interface SendHandler . . . 29
3-13 Java: Encoder Interface . . . 29
3-14 Java: Decoder Interface . . . 30
3-15 Java: Angabe der Encoder- und Decoder-Klassen . . . 30
3-16 Java: EndpointConfig . . . 31
3-17 Java: ClientEndpointConfig Konfigurationsklasse . . . 32
3-18 Java: ServerEndpointConfig Konfigurationsklasse . . . 33