Keine Einbahnstraße

WebSockets in Java EE 7

Rüdiger Grammes, Markus Günther, Martin Lehmann

Das WebSocket-Protokoll ist eine leichtgewichtige Alternative zu HTTP und ermöglicht die bidirektionale Kommunikation zwischen einem Client und einem Server. Die neue Java-Enterprise-Edition Java EE 7 enthält mit JSR 356 eine standardisierte Java-Bibliothek für WebSocket-Clients und -Server, die der Artikel vorstellt.

Warum überhaupt WebSockets?

HTTP ist ein Halb-Duplex-Protokoll, bei dem die Kommunikation von Server zu Client immer durch eine Client-Anfrage initiiert wird. Wartet ein Client auf Daten eines Servers, so muss er über das HTTP-Protokoll in regelmäßigen Abständen Anfragen an den Server schicken (Polling). Dies führt zu erhöhter Netzlast und zu zusätzlicher Latenz, schlimmstenfalls in der Länge des Polling-Intervalls. Je kürzer dieses Intervall gewählt wird, desto höher die erzeugte Last. Dabei enthält jede Polling-Anfrage einen kompletten HTTP-Request mit allen Header-Daten.

Um dem entgegenzuwirken, wurden eine Reihe von „Hacks" auf Basis des HTTP-Protokolls eingeführt, die unter dem Oberbegriff „Comet" bekannt sind. Diese lassen sich prinzipiell in Long-Polling- und Streaming-Techniken unterteilen. Beim Long-Polling schickt der Server bei einer Anfrage nicht direkt eine Antwort zurück, sondern wartet, bis Daten vorliegen oder bis ein Timeout eintritt. Das Polling-Intervall wird somit erheblich verlängert. Streaming-Techniken nutzen Eigenschaften von IFrames oder XMLHttpRequests, um über eine persistente Verbindung Events vom Server an den Client zu schicken.

WebSocket ist durch die IETF mit RFC 6455 seit Ende 2011 standardisiert [IETF11]. Dazu hat die W3C eine standardisierte Programmierschnittstelle für die bidirektionale Kommunikation zwischen Server und HTML5/JavaScript-Client im Browser festgelegt [W3C09]. Die WebSocket-Kommunikation basiert auf TCP, denn HTTP wird nur für den initialen Aufbau der Verbindung zwischen Client und Server verwendet. Der HTTP-Protokoll-Overhead entfällt danach. Die Stärken des WebSocket-Protokolls kommen insbesondere dann zum Tragen, wenn viele kleine Nachrichten (Text oder Binär) zwischen Client und Server ausgetauscht werden müssen, oder wenn der Client besonders schnell auf Events des Servers reagieren muss. Klassische Beispiele dafür sind Chat-Anwendungen oder Multiplayer-Spiele (s. auch [Stack]). Weitere Details zu WebSockets zeigt [Ron12].

Erlaubt die Anwendung große Latenzzeiten, so passt in der Regel ein klassischer Polling-Ansatz besser als WebSockets und ist einfacher zu realisieren. Wird keine bidirektionale Kommunikation benötigt, so sind Server Sent Events (SSE) eine HTML5-basierte Alternative zu WebSockets. SSE erlauben es dem Server, Nachrichten an den Client (als `text/event-stream`) über eine zuvor aufgebaute persistente HTTP-Verbindung zu senden (vergleichbar mit Comets Streaming-Ansätzen). Im Gegensatz zu WebSockets versucht SSE bei einem Verbindungsabbruch automatisch einen erneuten Verbindungsaufbau. Nachteile dabei: SSE unterstützt keine Domänen übergreifende Kommunikation und wird außerdem nicht von allen Browsern wie IE10 unterstützt. Weitergehende Vergleiche der Alternativtechnologien finden sich bei [LuGr13].

WebSocket-Kommunikation

Der Aufbau einer WebSocket-Verbindung erfolgt als Two-Way-Handshake über das HTTP/1.1-Protokoll. Der Client initiiert die WebSocket-Verbindung, indem er einen GET-Request mit den Header-Feldern `Upgrade: websocket` und `Connection: Upgrade` an den Server schickt. Zusätzlich muss er die verwendete Protokollversion (`Sec-WebSocket-Version`) und eine Base64-codierte Zufallszahl (`Sec-WebSocket-Key`) angeben. Browser-Clients sind verpflichtet, ihren Aufrufkontext im `Origin`-Feld mitzugeben. Optional kann der Client eine Liste von Subprotokollen (`Sec-WebSocket-Protocol`) angeben, über die er mit dem Server kommunizieren will. Subprotokolle sind außerhalb des WebSocket-Standards definierte Protokollerweiterungen, vgl. auch [IANA]. Ein Beispiel für `chat` und `superchat`:

```
GET /chat HTTP/10.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

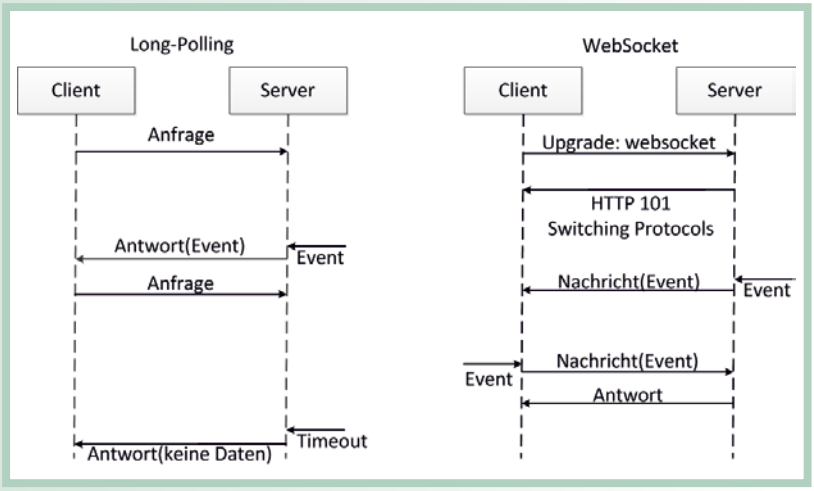

Abb. 1: Long-Polling und WebSocket im Vergleich

Ein Server nimmt die WebSocket-Verbindung mit dem Statuscode `101 Switching Protocols` an. Die Antwort enthält einen auf Basis des `Sec-WebSocket-Key` berechneten Schlüssel (`Sec-WebSocket-Accept`) und eine Teilmenge der angefragten Subprotokolle:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Nach erfolgreichem Verbindungsaufbau können beide Parteien über das WebSocket-Protokoll bidirektional kommunizieren, also Nachrichten senden und empfangen und den Kommunikationskanal schließen. Die Unterscheidung in Client und Server ist nach Aufbau der Verbindung hinfällig, beide sind danach gleichberechtigte Peers.

JSR 356 und Tyrus

Ende Mai 2013 ist das Final Release des Java EE 7-Standards verabschiedet worden. Eine der vier neuen Java EE-Spezifikationen ist die Unterstützung von WebSockets im JSR 356 [JSR356]. Die Open-Source-Referenzimplementierung heißt Tyrus [Tyrus]. Der Fokus des JSR liegt auf server- und clientseitiger Programmierschnittstelle sowie auf der Integration in die Java EE-Security.

Bei der Definition von server- und clientseitigen Endpoints werden zwei Programmierstile unterstützt: Endpoints können entweder programmatisch (Ableitung von API-Oberklassen bzw. Implementierung von Interfaces) oder über Annotationen (an Methoden eines POJO) deklariert werden.

```
 1: package echo.server;
 2: import javax.websocket.*;
 3: import javax.websocket.server.*;
 4:
 5: @ServerEndpoint(value = "/websockets/wsecho")
 6: public class EchoServer {
 7:   // serverID of this EchoServer instance
 8:   private String serverID = this.toString();
 9:
10:   // number of message received (in this EchoServer instance)
11:   private int messageCounter = 0;
12:
13:   @OnMessage
14:   public String doEcho(String textMessage, Session session) {
15:     System.out.println("Text message received = '" + textMessage);
16:     return "Response from " + serverID + " to call #" +
17:     (++messageCounter) + ". Text message received = " + textMessage;
18:   }
19:   @OnMessage
20:   public String doEcho(byte[] binMessage) {
21:     System.out.println(
          "Bin. message received = " + new String(binMessage));
22:     return "Response from " + serverID + " to call #" +
23:     (++messageCounter) + ". Binary message received = " + binMessage;
24:   }
25: }
```

Listing 1: Annotation-basierte Variante eines einfachen EchoServer-Endpoints

Das Beispiel in Listing 1 zeigt die Annotation-Variante zur Definition eines einfachen **`EchoServer`**. Die POJO-Klasse **`EchoServer`** wird über **`@ServerEndpoint`** als serverseitiger WebSocket-Server-Endpoint deklariert. Die beiden **`doEcho`**-Methoden sind mit **`@OnMessage`** annotiert; sie werden aufgerufen, wenn der Server-Endpoint eine Text- bzw. Binär-Nachricht erhält (**`String`** bzw. **`byte[]`**, alternativ **`ByteBuffer`** oder beliebiger Typ, für den ein Decoder existiert). Auch Pong-Nachrichten werden unterstützt (im Beispiel nicht gezeigt).

Eine Antwort kann verschickt werden, wenn sie mit **`return`** zurückgegeben wird (im Beispiel ein **`String`**, alternativ **`byte[]`** oder **`ByteBuffer`** oder beliebiger Typ, für den ein Encoder existiert). Alternativ kann die Methode auch **`void`** sein, sodass keine Antwort zurückgegeben wird.

Einen solchen Endpoint deployt man typischerweise innerhalb einer Webapplikation auf einen Java EE 7-konformen Applikationsserver wie beispielsweise Glassfish 4.0. Dieser Applikationsserver bringt eine Tyrus-Laufzeitumgebung mit.

Der HTML/JavaScript-Webclient aus Listing 2 kann nun bereits im Browser den **`EchoServer`** ansprechen. Der JavaScript-Code aus Listing 2 gibt alle mit dem **`EchoServer`** ausgetauschten Nachrichten direkt auf der HTML-Seite im Browser aus.

```
 1: var wsUri = "ws://localhost:8080/echoserver/websockets/wsecho";
 2: var websocket = new WebSocket(wsUri);
 3: websocket.onopen =
      function(event) { print("[CONNECTED] to: " + wsUri); };
 4: websocket.onmessage =
      function(event) { print("[RECEIVED] " + event.data); };
 5: websocket.onerror =
      function(event) { print("[ERROR] " + event.data); };
 6: function print(message) {
 7:   var pre = document.createElement("p");
 8:   pre.style.wordWrap = "break-word";
 9:   pre.innerHTML = message;
10:   document.getElementById("output").appendChild(pre);
11: }
12: function sendEcho() {
13:   websocket.send (messageID.value);
14:   print("[SENT] " + messageID.value);
15: }
```

Listing 2: Beispielhafter WebSocket-Client mit JavaScript

Die Browserunterstützung für WebSocket ist inzwischen sehr gut [CanIUse]. Müssen auch ältere Browser, die keine oder nur eine veraltete Version des Standards kennen, unterstützt werden, so muss ein Fallback auf eine andere Technologie, wie Long-Polling, implementiert werden. Im Gegensatz zu Java-Frameworks wie Atmosphere [Atmosphere] sind Fallback-Lösungen nicht Teil von Tyrus.

Beim Debugging leistet der Chrome-Browser gute Dienste: Mit seinen Developer-Tools sowie über **`chrome://net-internals`** kann man den WebSocket-Netzwerkverkehr detailliert ausgeben und mitschneiden.

Serverseitiges API des JSR 356

Wir haben bereits gezeigt, dass eine mit **`@OnMessage`** annotierte Methode im Server-Endpoint durch den WebSocket-Container aufgerufen wird, sobald eine Message eingeht. Weitere Annotationen aus dem Package **`javax.websocket`** legen den Lebenszyklus eines Server-Endpoints fest:

- **`@OnOpen`**: Die Methode wird beim Öffnen der WebSocket-Connection gerufen.
- **`@OnError`**: Die Methode wird gerufen, wenn ein Fehler auftritt.
- **`@OnClose`**: Die Methode wird beim Schließen der WebSocket-Connection gerufen.

Den Methoden kann optional eine **`Session`** als Parameter mitgegeben werden, über die Details der Connection erfragt werden können. Auch der Peer kann so ermittelt werden: Mit **`Session.getBasicRemote()`** bzw. **`Session.getAsyncRemote()`** erfolgt der Zugriff auf **`RemoteEndpoint.Basic`** (bei blockierendem Nachrichtenversand) und **`RemoteEndpoint.Async`** (nicht-blockierend).

Jede Instanz eines Server-Endpoints ist mit *genau einer* Connection und *genau einem* Peer assoziiert. Es ist aber auch möglich, *alle* aktuell verbundenen Peers mit **`Session.getOpenSessions()`** zu ermitteln. Dies ist nützlich, wenn Broadcast-Nachrichten zu versenden sind (s. Listing 3).

```
1: @OnMessage
2: public void broadcast(String message, Session session)
       throws IOException {
3:   for (Session receiver : session.getOpenSessions()) {
4:       receiver.getBasicRemote().sendText(message);
5:   }
6: }
```

Listing 3: Broadcast einer Nachricht

Clientseitiges API des JSR 356

Der JSR 356 spezifiziert auch die Programmierschnittstelle für einen Java-Client-Endpoint. So können auch Java SE-Anwen-

dungen mit einem WebSocket-Server wie unserem `EchoServer` kommunizieren (s. Listing 4).

```
 1: @ClientEndpoint
 2: public class MyClientEndpoint {
 3:   private static final String wsUri =
 4:     "ws://localhost:8080/echoserver/websockets/wsecho";
 5:   private static final int stopAfterNumMessage = 2;
 6:   private final static CountDownLatch messageLatch =
 7:     new CountDownLatch(stopAfterNumMessage);
 8:   @OnOpen
 9:   public void onOpen(Session session) {
10:     System.out.println("[CONNECT] to: " + session.getBasicRemote());
11:     try {
12:       String msg = "Hello WebSocket!";
13:       // sending text message to server
14:       session.getBasicRemote().sendText(msg);
15:       // sending binary message to server
16:       session.getBasicRemote().sendBinary(
            ByteBuffer.wrap(msg.getBytes()));
17:     } catch (IOException ex) {
18:       ex.printStackTrace(System.err);
19:     }
20:   }
21:   @OnMessage
22:   public void onMessage(String message) {
23:     System.out.println("[RECEIVED] " + message);
24:     // count down to 0, so that native client can finally exit ...
25:     messageLatch.countDown();
27:   }
28:   public static void main(String[] args) {
29:     try {
30:       WebSocketContainer container =
            ContainerProvider.getWebSocketContainer();
31:       container.connectToServer(
            MyClientEndpoint.class, URI.create(wsUri));
32:       // force current thread to wait until the latch
          // has counted down to 0
33:       messageLatch.await(60, TimeUnit.SECONDS);
34:     } catch (DeploymentException | InterruptedException |
            IOException ex) {
35:       ex.printStackTrace(System.err);
36:     }
37:   }
38: }
```

Listing 4: Annotation-basierte Variante eines WebSocket-Clients

Die Client-Annotationen `@OnOpen`, `@OnMessage` sowie der Versand beider Nachrichten am Ende von `onOpen()` sind analog zum Server-Endpoint. Einzige Besonderheit im Client-Code ist der `CountDownLatch`, der dafür sorgt, dass der native Client nicht terminiert, bevor zwei Nachrichten als Antworten auf die verschickten Nachrichten empfangen wurden.

Datenpakete und Datenarten

JSR 356 unterstützt unterschiedliche Verarbeitungskonzepte für eingehende Nachrichten, wobei unterschieden wird:

- Datenpakete: Lieferung der Nachricht partiell oder komplett?
- Datenart: textuelle oder binäre Nachricht?

Ein Empfänger kann entweder an einzelnen Datenpaketen (sog. „Chunks“, die erst gemeinsam eine vollständige Nachricht beschreiben) oder nur an einer vollständigen Nachricht interessiert sein. Insbesondere für Streaming-Applikationen ist die Möglichkeit interessant, auf Ebene einzelner Chunks informiert zu werden. Text-gestützte Applikationsprotokolle machen davon in der Regel keinen Gebrauch, da meist die vollständige Nachricht für die weitere Verarbeitung benötigt wird.

Die Programmierschnittstelle reflektiert die Konzepte sowohl programmatisch über spezielle Ausprägungen des Interfaces `MessageHandler` als auch über die `@OnMessage`-Annotation an einer Methode, die somit als Handler für den Nachrichtentyp fungiert.

Betrachten wir zunächst die programmatische Variante: Das Interface `MessageHandler` kennt zwei Spezialisierungen. Eine Realisierung von `MessageHandler.Whole<T>` erhält Benachrichtigungen über eingehende Nachrichten, die bereits in vollem Umfang vorliegen. Auch wenn eine Nachricht in Chunks aufgeteilt ist, speichert Tyrus die Einzelpakete im Cache zwischen, bis *alle* Datenpakete einer Nachricht vorliegen, und benachrichtigt erst dann den `MessageHandler`. Eine Realisierung von `MessageHandler.Partial<T>` wird dagegen über den Empfang einzelner Datenpakete benachrichtigt.

Der Typ `T` entscheidet über Text- oder Binär-Nachricht anhand der in Tabelle 1 angegebenen Zuordnung.

Nachricht	Mögliche Java-Typen

Text-Nachricht	`java.lang.String`
	`java.io.Reader`
	Typ `T`, für den eine Implementierung existiert von
	▼ `javax.websocket.Decoder.Text` oder
	▼ `javax.websocket.Decoder.TextStream`

Binär-Nachricht	`java.nio.ByteBuffer`
	`java.nio.InputStream`
	Typ `T`, für den eine Implementierung existiert von
	▼ `javax.websocket.Decoder.Binary` oder
	▼ `javax.websocket.Decoder.BinaryStream`

Tabelle 1: Zuordnung von Java-Typen zu Nachrichtenarten

Die Spezifikation hat die Restriktion, dass für eine Datenart nur ein `MessageHandler` registriert werden darf. Für die Verarbeitung von Textdaten kommen in Listing 5 die beiden registrierten `MessageHandler` in Frage.

```
1: @Override
2: public void onOpen(Session session, EndpointConfig endpointConfig) {
3:   session.addMessageHandler(
       new MessageHandler.Whole<String>() { ... } );
4:   session.addMessageHandler(
       new MessageHandler.Whole<Reader>() { ... } );
5: }
```

Listing 5: Registrierung textbasierter MessageHandler

Die programmatische Variante in Glassfish 4.0 mit Tyrus 1.0 lässt eine Registrierung mehrerer `MessageHandler` auf eine Datenart zu, wählt dann aber zur Laufzeit den zuerst registrierten `MessageHandler` aus, der für diese Datenart in Frage kommt – in unserem Beispiel also die Instanz von `MessageHandler.Whole<String>`. Dies widerspricht der Spezifikation des JSR 356, die fordert, dass in diesem Fall ein Laufzeitfehler geworfen werden muss. Bei der Variante mit Annotationen wird dagegen spezifikationsgerecht ein Fehler beim Deployment geworfen.

Implementierung von Anwendungsprotokollen

Sowohl Text- als auch Binärdaten können durch eigene `Decoder/Encoder` in Java-Typen konvertiert werden. Das ist sinnvoll, wenn Peers über WebSocket-Verbindungen unterschiedliche

Nachrichten austauschen, die in ihrer Gesamtheit ein Applikationsprotokoll repräsentieren:

- Ein `Encoder` erledigt die Serialisierung eines Java-Typs in eine textuelle oder binäre Repräsentation, die über die WebSocket-Verbindung gesendet werden kann.
- Ein `Decoder` konvertiert empfangene Daten in eine Instanz eines Java-Typs.

Ein `Decoder`/`Encoder`-Paar realisiert zusammen eine Art Codec. Die API-Interfaces differenzieren sich aus in `Decoder.Text<T>`, `Decoder.TextStream<T>`, `Decoder.Binary<T>` und `Decoder.BinaryStream<T>` (analog für `Encoder`). `T` stellt den Java-Typ dar, der die deserialisierte Nachricht repräsentiert.

Für unser Echo-Beispiel implementieren wir einen Codec für eine `EchoMessage`, die die übertragene Zeichenkette kapselt und getter-/setter-Methoden bereitstellt. Ein `EchoMessageEncoder` codiert in der Methode `encode()` Instanzen von `EchoMessage` in ausgehende Textdaten (s. Listing 6).

```
1: public class EchoMessageEncoder
       implements Encoder.Text<EchoMessage> {
2:   @Override
3:   public String encode(EchoMessage echoMessage)
         throws EncodeException {
4:     return echoMessage.getMessage();
5:   }
     ...
```

Listing 6: Encoder-Implementierung zur Serialisierung von EchoMessage-Objekten

```
1: public class EchoMessageDecoder
       implements Decoder.Text<EchoMessage> {
2:   @Override
3:   public EchoMessage decode(final String s) throws DecodeException {
4:     return new EchoMessage(s);
5:   }
6:   @Override
7:   public boolean willDecode(final String s) {
8:     return s != null;
9:   }
     ...
```

Listing 7: Decoder-Implementierung zur Deserialisierung von EchoMessage-Objekten

Den zugehörigen `EchoMessageDecoder` zeigt Listing 7. Leider unterstützt JSR 356 aktuell nicht, mehrere `Decoder` auf einer Datenart zu definieren. Dies stellt bei der Implementierung komplexerer Applikationsprotokolle eine kritische und unnötige Restriktion dar. Möchte man eingehende Daten in mehrere Nachrichtenobjekte auf Java-Seite decodieren, so kann man sich derzeit nur mit einem generischen Objekt behelfen und muss die Delegation an entsprechende Handler-Methoden selbst implementieren.

Fazit

Mit WebSockets können auf einfache Art und Weise interaktive Client-Server-Kommunikationen im Web implementiert werden, die weit über das hinausgehen, was HTTP alleine und ohne weitere Tricks zu leisten imstande ist. Mit Java EE 7 ist der IETF-Standard nun auch in der Java-Enterprise-Welt verfüg- und einsetzbar. Das JSR-356-API ist sauber definiert und „fühlt sich gut an“. Ausführliche Dokumente zu API und Tyrus sind neben einführenden Beispielen [JEE 7 Samples] verfügbar. Aber die beschriebenen Restriktionen für `MessageHandler` und `Decoder`/`Encoder` erfordern noch mühsame Abstraktionsarbeit, sobald man mehr als nur Hello-World-Beispiele implementiert. In realen Projekten sind viele Datentypen und viele Client-Server-Schnittstellen der Normalfall, sodass diese Projekte wohl zuerst einige Basisklassen und Abstraktionen einführen müssen, um die Restriktionen wegzukapseln. Dies hätte man sich bereits im aktuellen JSR 356 gewünscht. Trotzdem machen WebSockets auch mit JSR 356 und Tyrus Spaß und werden sicherlich ihre Anhänger finden.

Literatur und Links

[Atmosphere] Getting Started with the Atmosphere Framework and WebSocket, `https://github.com/Atmosphere/atmosphere/wiki/Getting-Started-with-The-Atmosphere-Framework-and-WebSocket`

[CanIUse] Can I Use WebSockets?, `http://caniuse.com/websockets`

[IANA] WebSocket Protocol Registries, Internet Assigned Numbers Authority, `https://www.iana.org/assignments/websocket/websocket.xml#subprotocol-name`

[IETF11] Internet Engineering Task Force (IETF) Request for Comments (RFC) 6455: The WebSocket Protocol, `https://tools.ietf.org/html/rfc6455`

[JEE7Samples] A. Gupta, JEE7 Samples Galore, `https://blogs.oracle.com/arungupta/entry/java_ee_7_samples_galore`

[JSR356] Java Specification Request 356: Java™ API for Web-Socket, `http://jcp.org/en/jsr/detail?id=356`

[LuGr13] P. Lubbers, F. Greco, HTML5 WebSockets: A Quantum Leap in Scalability for the Web, `http://www.websocket.org/quantum.html`

[Ron12] A. Ronacher, WebSockets 101, `http://lucumr.pocoo.org/2012/9/24/websockets-101`

[Stack] WebSocket application that is not a game, chat, twitter client or market index, `http://stackoverflow.com/questions/9363894/websocket-application-that-is-not-a-game-chat-twitter-client-or-market-index`

[Tyrus] `http://tyrus.java.net/`

[W3C09] W3C, The Web Sockets API, `http://www.w3.org/TR/2009/WD-websockets-20091222/`

Rüdiger Grammes ist seit November 2011 als Managing Consultant bei der Accelerated Solutions GmbH und dort als Softwarearchitekt in verschiedenen Projekten unterwegs.
E-Mail: grammes@accso.de

Markus Günther ist als Senior Software Engineer bei der Accelerated Solutions GmbH tätig und beschäftigt sich dort vornehmlich mit der Konzeption und Entwicklung Java-basierter Systeme.
E-Mail: guenther@accso.de

Martin Lehmann ist seit September 2010 als Cheftechnologe und Softwarearchitekt bei der Accelerated Solutions GmbH tätig.
E-Mail: lehmann@accso.de