31.8.–3.9.2015
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Rufen Sie uns nicht an, wir rufen Sie an!

Server-Sent Events und WebSockets im Java Enterprise Umfeld

Sebastian Reiners

open knowledge GmbH

Inhalt

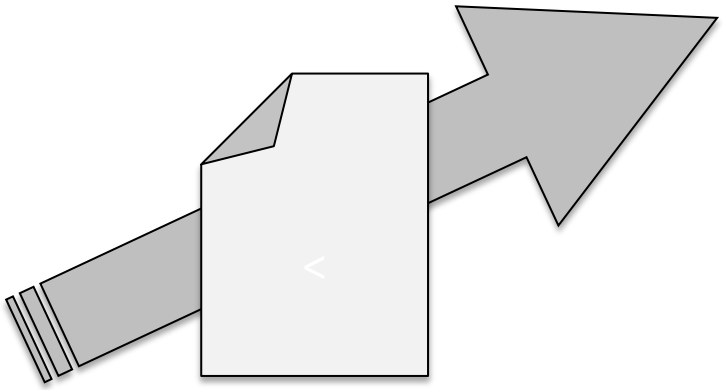
1. HTTP: Wer nicht fragt bleibt dumm
2. Alternative Ansätze
3. Server-Sent Events
4. WebSockets
5. Probleme beim Einsatz mit JavaEE
6. Fazit

Funktionsweise von HTTP

- Klassische Funktionsweise des Web:
  1. Client stellt Anfrage an Server <= Request
  2. Server verarbeitet Anfrage
  3. Server schickt Antwort <= Response
  4. Client verarbeitet Antwort
- **Motto:** Wer nicht fragt bleibt dumm.

Funktionsweise von HTTP

- Standard-Verhalten, seit es HTTP gibt (seit 1991)
- Dokumenten-basiert
  - ein Request => ein neues Dokument
- zugeschnitten auf damaligen Verwendungszweck
  - Austausch von Dokumenten



Es kommt Bewegung rein

- sehr starres Konstrukt
- Dynamik? ☹
- Wechsel von Dokumenten zu Anwendungen im Web
  - zunächst mit synchroner Verarbeitung
- nächster, großer Schritt: Auflockern des Request/Response Cycles
  - unterstützt durch clientseitige Skript-Sprache
- Asynchronous JavaScript and XML

Asynchrone Requests

- Auflockern, noch kein Auf**brechen**
- lösen vom dokumenten-basierten Ansatz
- Übertragung von
  - ... Teildokumenten
  - ... dokumentenunabhängiger Information
- 1998 durch Microsoft angestoßen worden
- 2006 durch W3C standardisiert als XMLHttpRequest

Asynchrone Requests

- Vorteile asynchroner Requests:
  - mehr Dynamik in Webseiten
  - geringeres Datenvolumen im Transport
  - Loslösung von reiner Formularverarbeitung
- Es ist aber immer noch das Request/Response Prinzip!

Asynchrone Requests

- in vielen Fällen immer noch unpraktisch
- Was ist bei Änderungen auf Serverseite?
- Änderungen können erst bei nächster Anfrage mitgeteilt werden.
  - zeitliche Verzögerung

Welche Ansätze gibt es?

- zeitlich getaktet, immer wieder beim Server anfragen
  - Wie kurz takte ich?
- über Applets
- Longpolling
  - Grundlage: langlaufende Requests
  - wiederholter, erneuter Aufbau

Welche Ansätze gibt es?

- Nachteile:
  - haben Sicherheitsprobleme
  - holprige Integration mit dem Rest der Seite
  - benötigen ein weiteres Protokoll, aufsetzend auf HTTP
  - Skalierungsprobleme

Wäre doch schön wenn ...

Server-Sent Events und WebSockets

- beides standardisiert durch das W3C
- beides in Java Webanwendungen einsetzbar
- Server-Sent Events: *unidirektional*
  - über eine HTTP Connection
- WebSockets: *bidirektional*
  - über eine Socket-Verbindung

Server-Sent Events

- Client "registriert" sich beim Server
- kann danach vom Server "Events" empfangen
- Event-Nachrichten haben ein bestimmtes Format:

```
event:add\n
data:1500\n
data:600\n
\n
```

- Übermittlung mittels HTTP

Server-Sent Events

- in vielerlei Hinsicht ähnlich zu Longpolling Verfahren
  - allerdings fehlender Ab-/Aufbau der Verbindung
- gut geeignet für beständige Aktualisierungen
  - Aktienkurse
  - Newsfeeds
  - wann immer der Client lange warten müsste

Server-Sent Events - Client

```
var source = new EventSource('/sse_xyz');
source.onmessage = function(event) {
      if (event.type === 'add') {
            ...
      } else if (event.type === 'remove') {
            ...
      }
}
...
source.close();
```

Server-Sent Events - Server

- Gibt es einen JSR für Server-Sent Events?
  - Nein, brauchen wir aber auch nicht. ☺
  - Die Standard Servlet API ist bereits vollkommen ausreichend.

Server-Sent Events - Server

```
@WebServlet(urlPatterns={"/sse_xyz"})
public class SSEServlet extends HttpServlet {

    @Override
    protected void doGet(   HttpServletRequest req,
                            HttpServletResponse res)
            throws IOException, ServletException {

              res.setContentType("text/event-stream");
              res.setCharacterEncoding("UTF-8");
              PrintWriter writer = res.getWriter();
              ...
              writer.write("data: " + msg + "\n\n");
              writer.flush();
    }
}
```

Server-Sent Events

- möglich mit jedem Servlet-Container

- keine Unterstützung durch IE / Edge
- bringen u.U. Skalierungsprobleme mit sich

WebSockets

- bidirektionale Kommunikation
  - sowohl Client als auch Server können selbstständig Nachrichten schicken
- Client "registriert" sich beim Server
  - Request über "ws://" bzw. "wss://"
  - Handshake über HTTP
  - Danach eigentlicher Protokollwechsel

WebSocket - Handshake

```
GET /chat/connect HTTP/1.1
Host: localhost:8080
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
Sec-WebSocket-Extensions: permessage-deflate
Sec-WebSocket-Key: M+Le93PunXLSl33Vpip98Q==
Connection: keep-alive, Upgrade
Upgrade: websocket
```

WebSocket - Handshake

```
HTTP/1.1 101 Switching Protocols

Origin: http://localhost:8080

Upgrade: WebSocket

Sec-WebSocket-Accept: zH6VNxS6o1H/X1ZRqv4a213BanQ=

Connection: Upgrade

Sec-WebSocket-Location: ws://localhost:8080/chat/connect

Content-Length: 0
```

WebSockets

- die mächtigste der hier vorgestellten Kommunikationsarten
- kann alles erfüllen, was über Server-Sent Events möglich ist
- hat einen zusätzlichen Kanal für Nachrichten des Clients
- gut geeignet für:
  - Chats
  - Collaboration-Tools

WebSockets – JavaScript Client

```
var socket = new WebSocket("ws://host:8080/path/");

socket.onopen = function(event) {...}
socket.onmessage = function(event) {
      alert("Received message: "+event.data);
}
socket.onclose = function(event) {...}

socket.send("Message-Inhalt");
...
socket.close();
```

WebSockets - Serverseite

- Gibt es dazu einen JSR? (Oder braucht man ihn hier auch nicht?)
- Ja, gibt es. JSR 356
  - Seit Java EE 7 verfügbar
- Server-Support:
  - Glassfish 4
  - Wildfly 8
  - Jetty 9.1
  - ...

WebSockets - Serverseite

- WebSocket Unterstützung schon vor JSR
- große Unterschiede in der Umsetzung
  - sehr rudimentär: Glassfish
  - sehr weit fortgeschritten Jetty
- Umsetzung des JSR => Jetty-Version +
  - Ein-/Ausgabe Konverter
  - Standardisierte Schnittstelle zum Versand von Text, Binärdaten, usw.

WebSockets - Serverseite

- Wie funktioniert das denn nun?
- Zwei Ansätze:
  - Interface-basiert
  - Annotation-basiert

WebSockets - Serverseite

- (nahezu) beliebige POJOs als Endpoint
- `@ServerEndpoint("/pathForEndpoint")`
- `@OnOpen` an Methoden für Verbindungsaufbau
- mit folgender Signatur:

```
public void method(Session session,
                   EndpointConfig cfg,
                   @PathParam(...) String param)
```

WebSockets - Serverseite

- `@OnClose` an Methoden für Verbindungsabbau
- mit folgender Signatur:

```
public void method(Session session,
                   CloseReason reason,
                   @PathParam(...) String param)
```

WebSockets - Serverseite

- `@OnMessage` an Methode, die auf Client-Nachrichten reagieren soll
- Signatur

```
public void method([Variiert] message,
                   Session session,
                   @PathParam(...) String param)
```

WebSockets - Serverseite

- Für Textnachrichten:
  - `String`, Java Primitives
  - `Reader`
- Für Binärnachrichten:
  - *byte[]*,
  - `ByteBuffer`

WebSockets – Ein einfaches Beispiel

- Genug Theorie ;-)
- Angenommen wir wollen folgenden Service schreiben:
  - Erreichbar unter

    ```
    ws://hostname:8080/chat/upper
    ```

  - Texte die hingeschickt werden, sollen Uppercase an alle Clients weitergeleitet werden, die verbunden sind

WebSockets – Ein einfaches Beispiel

```
@ServerEndpoint("/upper")
public class MySocketEndpoint {

  @OnOpen
  public void open(Session newSession) {
    System.out.println("Open:"+newSession.getId());
  }

  @OnClose
  public void close(Session closedSession) {
    System.out.println("Close:"+closedSession.getId());
  }
```

WebSockets – Ein einfaches Beispiel

```
  @OnMessage
  public void sendUpper(String message, Session se) {

    for(Session session : se.getOpenSessions()) {
      try {
        session.getBasicRemote()
               .sendText(message.toUpperCase());
      } catch (IOException | EncodeException e) {
        // Error handling
      }
    }
  }
}
```

WebSockets – Abseits von Text und Bytes

- Was wenn ich nicht auf Text- oder Binärdaten arbeiten will?
- Decoder und Encoder um Ein- und Ausgabedaten in ein eigenes Objektmodell umzuwandeln
- Bereitgestellte Interfaces:
  - `Decoder.Text, Decoder.Binary`
  - `Encoder.Text, Encoder.Binary`

Abseits von Text und Bytes - Encoder

```
public interface Encoder {
  void init(EndpointConfig var);
  void destroy();

  [...]

  public interface Text<T> extends Encoder {
    String encode(T var) throws EncodeException;
  }
}
```

Abseits von Text und Bytes - Decoder

```
public interface Decoder {
  void init(EndpointConfig var);
  void destroy();

  [...]

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

WebSockets – Abseits von Text und Bytes

```
public class Converter implements Decoder.Text<Message>,
                                  Encoder.Text<Message> {
        public void destroy() {...}
        public void init(EndpointConfiguration conf) {...}

        public Message decode(String messageStr) {...}
        public String encode(Message message) {...}

        public boolean willDecode(String messageStr) {...}
}
```

WebSockets – Abseits von Text und Bytes

- Wie geschieht die Anbindung an den Endpoint?

```
@ServerEndpoint(value="/endpoint",
                encoders=Converter.class,
                decoders=Converter.class)

@OnMessage
public void handleMessage(Message m, Session s) {
  for(Session session : s.getOpenSessions()) {
      session.getBasicRemote().sendObject(m);
  }
}
```

WebSockets – Ist das schon alles?

- Streaming-Unterstützung
  - `Encoder.TextStream`, `Decoder.TextStream`
    - `Reader` bzw. `Writer`
  - `Encoder.BinaryStream`, `Decoder.BinaryStream`
    - `InputStream` bzw. `OutputStream`

WebSockets – Ist das schon alles?

- Pfad-Parameter
  - ähnlich wie man es aus REST APIs gewohnt ist

```
@ServerEndpoint("path/{param}/endpoint")


@OnMessage
public void onMessage(String s,
      @PathParam("param") String par,
      Session session) {...}
```

WebSockets – Ist das schon alles?

- Java-Client:

```
@ClientEndpoint
public class WebsocketClientEndpoint {
  private MessageHandler messageHandler;
  public WebsocketClientEndpoint(URI endpointURI) {
    try {
      WebSocketContainer container =
        ContainerProvider.getWebSocketContainer();
      container.connectToServer(this, endpointURI);
    } catch (Exception e) {...}
  }
```

Einsetzbarkeit im Java EE Bereich

Okay ... Möglich ist das alles ...

Aber geht das auch wirklich gut?

Einsetzbarkeit im Java EE Bereich

- Server-Sent Events:
  - ältere, bewährte Technologie-Basis
  - Technologie immer noch Grundlage vieler neuer APIs
  - keine letztendliche Integration mit JSF
  - keine Unterstützung durch Internet Explorer
    - auch nicht in neuen Versionen

Einsetzbarkeit im Java EE Bereich

- WebSockets:
  - neuer JSR
  - Unterstützung durch alle **neueren** Browser
    - ggf. Alternative zu SSEs für Internet Explorer
  - keine letztendliche Integration mit JSF
  - unvollständige Integration mit CDI

Wo ist das Problem mit CDI?

```
@ServerEndpoint("/ws_connect")
public class MyEndpoint {

    private MySessionObject mso;

    ...
```

org.jboss.weld.context.ContextNotActiveException:
WELD-001303 No active contexts for scope
type javax.enterprise.context.SessionScoped

Wo ist das Problem mit CDI?

- Warum nicht aktiv? Wir haben doch eine Session ... Ô_ó
- Ja, aber ...
  - im Web-Container gebunden an HTTP-Session
  - WebSockets haben eigene Sessions, da nicht mehr HTTP
- Komme ich irgendwie an die HTTP-Session ran?
  - Ja, wir haben ja den anfänglichen Handshake!

Die ServerEndpoint Konfiguration

```
public class SessionAccessConfig extends
ServerEndpointConfig.Configurator {
  @Override
  public void modifyHandshake(ServerEndpointConfig cfg,
                              HandshakeRequest request,
                               HandshakeResponse response) {
    HttpSession session = (HttpSession) request.getHttpSession();
    cfg.getUserProperties().put("httpSession", session);
  }
}
```

Die ServerEndpoint Konfiguration

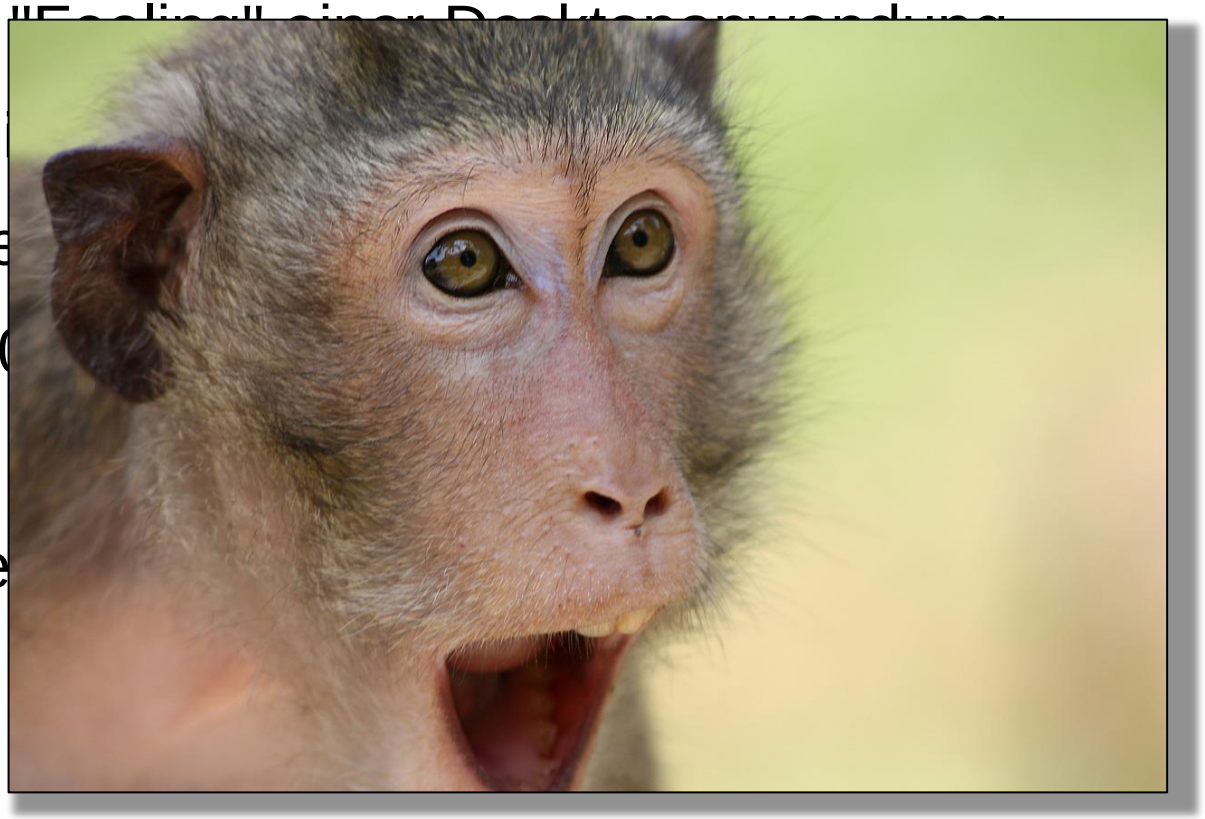
```
@ServerEndpoint(value="/upper",
                configurator=SessionAccesConfig.class)
public class MySocketEndpoint {
  private String valueFromHttpSession = null;

  @OnOpen
  public void open(Session newSession, EndpointConfig cfg) {
    HttpSession httpSes = (HttpSession)cfg.getUserProperties
                                .get("httpSession");
    valueFromHttpSession = httpSes.getAttribute("xyz");

  }
  ...
}
```

Was bringt mir das alles?

- stärkeres "Feeling" einer Desktopanwendung
- zunächst
  - höher, je
  - bis zu 4
- relativiert
  - Netzlate



Fazit

- WebSockets und Server-Sent Events in modernen JavaEE Umgebungen einsetzbar
- schwer mit JSF und CDI integrierbar
  - problemlos für kleine, losgelöste Features
  - als Anwendungs-Grundlage: Umdenken erforderlich

Fazit

- beides wird nicht "die Welt verändern"
- Server-Sent Events:
  - mangelnde Unterstützung durch IE
- WebSockets:
  - häufig wird man gerade auf CDI nicht verzichten wollen
  - Low-Level Protokoll

Fazit

- sind eine interessante Ergänzung
- läuft neben klassischen Enterprise Anwendungen
- wenn es auf Geschwindigkeit ankommt unschätzbar
- sorgen für einen intuitiveren Ablauf von Kommunikation

"Rufen Sie nicht den Server an, der Server ruft Sie an!"

Fragen?

Vielen Dank

Sebastian Reiners

open knowledge **GmbH**

www.openknowledge.de

facebook.com/openknowledge

twitter.com/_openknowledge