

# Java EE 7 and WebSocket API for Java (JSR 356) with AngularJS on WildFly

Maxime Greau

Revision 1.0

November 6, 2013

---

# Table of Contents

.....	v
1. Java EE 7 overview .....	1
2. DEMO : AngularJS - HTML5 / JSR-356 API application deployed on WildFly 8 (OpenShift) .....	2
3. WebSocket (WS) : a new protocol different from HTTP .....	4
3.1. Opening Handshake .....	6
3.2. Data transfer .....	7
3.3. Closing Handshake .....	7
4. WebSocket Javascript API (Client) .....	8
5. JSR 386 : Java API for WebSocket protocol .....	10
5.1. WebSocket Server Endpoint .....	10
5.2. Annotations .....	11
5.3. Encoders and Decoders .....	12
5.4. WebSocket Client Endpoint .....	12
6. US OPEN Application .....	13
6.1. Maven dependencies for Java EE 7 API .....	14
6.2. Add Server Endpoint .....	15
6.3. Encodes and Decodes messages .....	18
6.4. HTML5 Web Client (single match - index.html) .....	18
6.5. AngularJS Client (several matches - matches.html) .....	20
6.6. Source code on Github .....	21
6.7. Build and Deploy the WAR .....	21
7. Benchmark : WebSocket VS REST .....	22
8. References about WebSocket .....	23
9. Conclusion .....	24

---

## List of Figures

1.1. The 3 goals of Java EE 7 .....	1
2.1. US Open Application - Implementation of WebSocket (Java API et Javascript API) .....	3
3.1. How does the WebSocket protocol work .....	5

---

## List of Examples

3.1. HTTP Handshake sample request .....	6
3.2. HTTP Handshake Response sample .....	6
4.1. Javascript source code example, from <a href="http://websocket.org">http://websocket.org</a> .....	8
5.1. Java Client Endpoint sample .....	12
6.1. Maven project structure .....	13
6.2. pom.xml with Java EE 7 dependencies .....	14
6.3. Server Endpoint : MatchEndpoint.java .....	15
6.4. Text Encoder : MatchMessageEncoder.java .....	18
6.5. API Javascript implemented into websocket.js .....	19



## Overview

This blog post describes the [Java API for WebSocket Protocol \(JSR 356\)](#)<sup>1</sup> (which is one of four newest JSRs for the [Java EE 7](#)<sup>2</sup> platform) and provides a concrete application deployed on [WildFly 8](#)<sup>3</sup> and [available online on OpenShift](#)<sup>4</sup>.

- [Download an English PDF version](#)<sup>5</sup>
- [FR] La version française ([HTML](#)<sup>6</sup> ou [PDF](#)<sup>7</sup>) de ce post est basée uniquement sur la démonstration avec l'API Javascript sans AngularJS.

Once you will have read this post, you will be able to understand [Arun Gupta](#)<sup>8</sup>'s definition about what is it possible to do with WebSocket technology.

---

```
"WebSocket gives you bidirectionnal, full duplex, communication channel  
over a single TCP."
```

```
-- Arun Gupta (Java EE Evangelist chez Oracle) - Devovx UK 2013
```

---

---

<sup>1</sup> <http://jcp.org/en/jsr/detail?id=356>

<sup>2</sup> <http://jcp.org/en/jsr/detail?id=342>

<sup>3</sup> <http://wildfly.org>

<sup>4</sup> <http://wildfly-mgreau.rhcloud.com/usopen/matches.html>

<sup>5</sup> <http://mgreau.com/doc/javaee7-api-websocket-html5-en.pdf>

<sup>6</sup> <http://mgreau.com/posts/2013/09/27/javaee7-api-websocket-html5.html>

<sup>7</sup> <http://mgreau.com/doc/javaee7-api-websocket-html5.pdf>

<sup>8</sup> <https://twitter.com/arungupta>

# Chapter 1. Java EE 7 overview

The **Java Platform Enterprise Edition** was released in Version 7 (Java EE 7) in **June 2013**. In line with the two previous versions (Java EE 5 and Java EE 6) **Java EE 7** always proposes to simplify the work of the developer. This version decorates previous versions with 3 main objectives :

- embraces **HTML5** (WebSocket API, JSON-P API, JAX-RS)
- provide an **even better productivity** to developer (JMS)
- meeting **enterprise demands** (Batch API, Concurrency Utilities)



**Figure 1.1. The 3 goals of Java EE 7**

Java Platform, Enterprise Edition 7 (JSR 342) can be summed up around :

- 4 newest specifications : `Java API for WebSocket 1.0`, `Java API for JSON Processing 1.0`, `Batch Applications 1.0` and `Concurrency Utilities for Java EE 1.0`
- 3 specifications with major updates : `JMS 2.0`, `JAX-RS 2.0` and `EL 3.0`
- and 6 specifications with minor updates : `JPA 2.1`, `Servlet 3.1`, `EJB 3.2`, `CDI 1.1`, `JSF 2.2` and `Bean Validation 1.1`

---

# Chapter 2. DEMO : AngularJS - HTML5 / JSR-356 API application deployed on WildFly 8 (OpenShift)

If you want to see right away what it looks like, you can access [the online application<sup>1</sup>](#) whose code will be in part explained in this article. It's an application that give you the ability :

- to watch one or several tennis matches in **live mode** (Quarter Final U.S. Open 2013)
  - click on each match that you want to access live
  - click on exit button to disconnect from a specific match
- to bet on the winner of the match
  - each time a user bet on the same match, the counter is incremented
  - at this end of the match, you will see the result of your bet

You will say: *"Nothing special !"*, and you're right :)

At first glance, it sounds like something already seen in many of today's applications, but it's the technique used behind which does matter because, as you will see below, everything is based around the **standard of the new WebSocket protocol (ws:// ou wss://)** and not on "HTTP hacking".



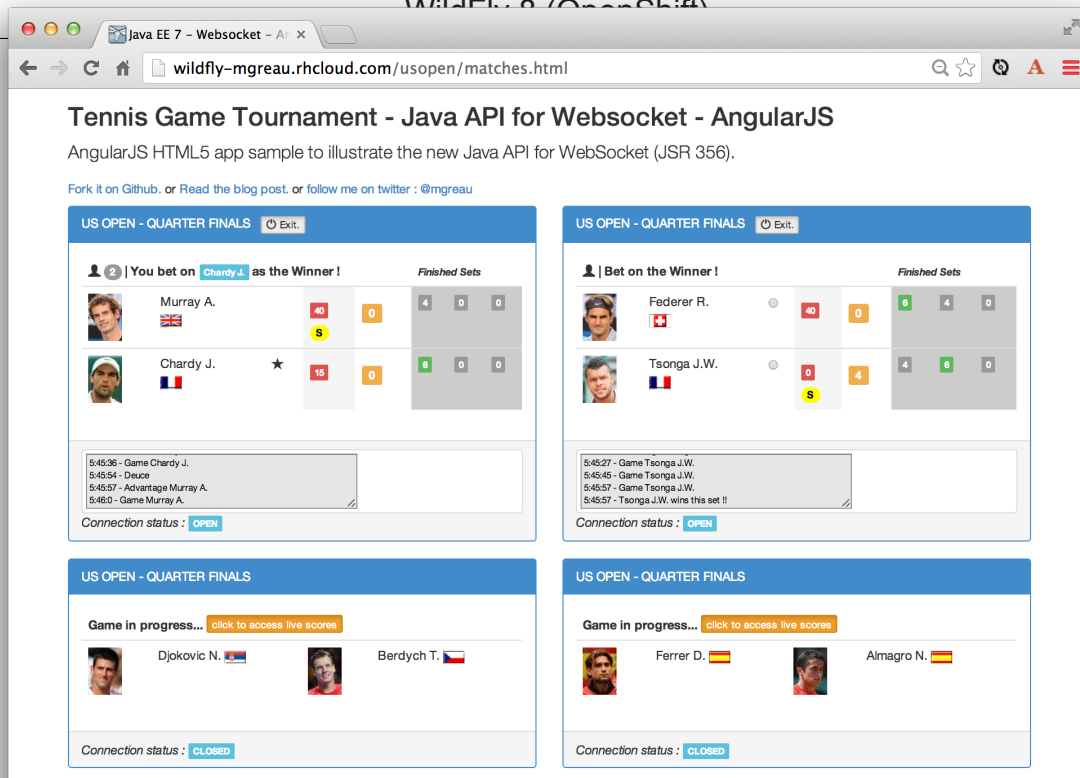
## The technologies used for the development of this application are :

- Frontend : `HTML5`, `CSS`, `Javascript` (`WebSocket API`) with *Bootstrap CSS* and *AngularJS*
- Backend : `Java API for WebSocket`, `EJB`, `JSON-P`, `JAX-RS`

---

<sup>1</sup> <http://wildfly-mgreau.rhcloud.com/usopen/matches.html>

DEMO : AngularJS -  
HTML5 / JSR-356 API  
application deployed on  
WildFly 8.0.0-Beta1 (OpenShift)



**Figure 2.1. US Open Application - Implementation of WebSocket (Java API et Javascript API)**

Nope! This demonstration is **not a chat application** :) It's obvious that the "chat demo" is the one that first comes to mind to illustrate the use of WebSocket technology. However, there are many other use cases, such as collaborative work on a text document online or online games like chess presented at the [JavaOne 2013 keynote](#)<sup>2</sup>.



*This application is available on the Cloud thanks to [OpenShift](#)<sup>3</sup>, the cloud computing PaaS product by RedHat. It's deployed on WildFly 8.0.0-Beta1 (normally certified Java EE 7 to the end of 2013). To set up an application server like WildFly on OpenShift, you just need to read [this Shekhar Gulati's blog post](#)<sup>4</sup>*

<sup>2</sup> [https://blogs.oracle.com/javaone/entry/the\\_javaone\\_2013\\_technical\\_keynote](https://blogs.oracle.com/javaone/entry/the_javaone_2013_technical_keynote)

<sup>3</sup> <https://www.openshift.com/>

<sup>4</sup> <https://www.openshift.com/blogs/deploy-websocket-web-applications-with-jboss-wildfly>



---

## Chapter 3. WebSocket (WS) : a new protocol different from HTTP

**HTTP**<sup>1</sup> is the standard protocol for the Web, it's very effective for a lot of use cases but, nevertheless, has **some drawbacks** in the case of **interactive Web applications** :

- **half-duplex** : based on the request/response pattern, the client sends a request and the server performs processing before sending a response, the client is forced to wait for a server response
- **verbose** : a lot of information are send in HTTP headers associated with the message, both in the HTTP request and in the HTTP response
- in order to add a **server push** mode, you need to use workaround (polling, long polling, Comet/Ajax) since there is no standard

This protocol is not optimized to scale on large applications that have significant needs of real-time bi-directional communication. This is why the **new WebSocket protocol** offers more advanced features than HTTP because it is:

- based on `1 unique TCP connection between 2 peers` (whereas each HTTP request/response needs a new TCP connection)
- `bidirectionnal`: client can send message to server and server can also send message to client
- `full-duplex`: client can send multiple messages to server, as well as server to client without waiting for a response from each other



*The term **client** is used only to define the one that initiate the connection. Once the connection is established, client and server become both **peers**, with the same capacity.*

The WebSocket protocol was originally intended to be part of the HTML5 specification but as HTML5 will be officially released in 2014, the WebSocket protocol is finally set, as well as HTTP protocol, by an IETF specification, [with RFC 6455](http://tools.ietf.org/html/rfc6455)<sup>2</sup>.

As shown in the diagram below, the **WebSocket protocol works in two phases** named :

---

<sup>1</sup> <http://tools.ietf.org/html/rfc2616>

<sup>2</sup> <http://tools.ietf.org/html/rfc6455>

1. `handshake` (open and close)
2. `data transfer`

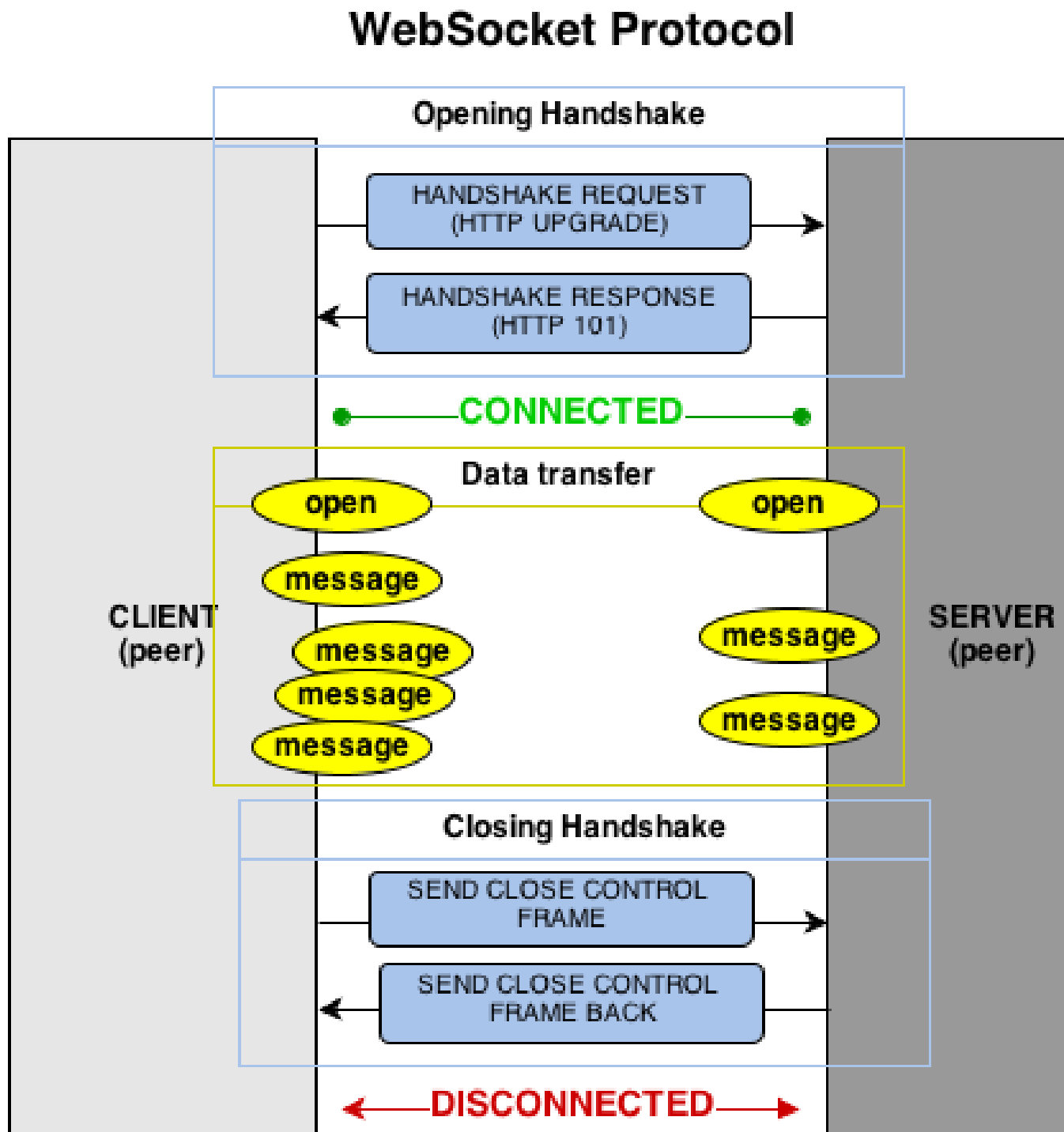


Figure 3.1. How does the WebSocket protocol work

## 3.1. Opening Handshake

The **Opening Handshake** phase is a **unique HTTP request/response** between the one who initiate the connection (peer client) and the peer server. This HTTP exchange is specific because it uses the concept of **Upgrade, defined in the HTTP specification<sup>3</sup>**. The principle is simple : **Upgrade HTTP** allows the client to ask the server to change the communication protocol and thus ensure that the client and server can discuss using a protocol other than HTTP.

### Example 3.1. HTTP Handshake sample request

```
GET /usopen/matches/1234 HTTP/1.1 ❶  
Host: wildfly-mgreau.rhcloud.com:8000 ❷  
Upgrade: websocket ❸  
Connection: Upgrade ❹  
Origin: http://wildfly-mgreau.rhcloud.com  
Sec-WebSocket-Key:0EK7XmpTZL341oOh7x1cDw==  
Sec-WebSocket-Version:13
```

- ❶ HTTP GET method and HTTP 1.1 version required
- ❷ Host used for the WebSocket connection
- ❸ Request to upgrade to the WebSocket protocol
- ❹ Request to upgrade from HTTP to another protocol

### Example 3.2. HTTP Handshake Response sample

```
HTTP/1.1 101 Switching Protocols ❶  
Connection:Upgrade  
Sec-WebSocket-Accept:SuQ5/hh0kStSr6oIzDG6gRfTx2I=  
Upgrade:websocket ❷
```

- ❶ HTTP Response Code 101 : server is compatible and accept to send messages through another protocol
- ❷ Upgrade to the WebSocket protocol is accepted

---

<sup>3</sup> <http://tools.ietf.org/html/rfc2616#section-14.42>



*When the upgrade request from HTTP to WebSocket protocol is approved by the endpoint server, it's no longer possible to use HTTP communication, all exchanges have to be made through the WebSocket protocol.*

## 3.2. Data transfer

Once the **handshake** is approved, the use of WebSocket protocol is established. There are an open connection on the *peer server side* as well on the *peer client side*, callback handlers are called to initiate the communication.

The **Data transfer** can now begin, so the 2 peers can exchange messages in a bidirectionnal and full-duplex communication.

As shown in the diagram named **Figure 3**, the `peer server` can send multiple messages (*in this example : 1 message for each scored point, 1 message each time any user bet on this game and 1 message at the end of the match*) without any `peer client` response and the peer client can also send messages at any time (*in this example : betting on the winner of the match*). Each peer can send a specific message to close the connection.

With Java EE7 Platform, the `peer server side` code is written in **Java** while the `peer client side` code is in **Java or Javascript**.

## 3.3. Closing Handshake

This phase **can be initiated by both peer**. A peer that want to close the communication need to send a **close control frame** and it will received a close control frame too as a response.

---

# Chapter 4. WebSocket Javascript API (Client)

To communicate from a Web application with a server using the WebSocket protocol, it's necessary to use a **client Javascript API**. It's the role of W3C to define this API. The W3C specification for the [JavaScript WebSocket API](http://w3.org/TR/websockets/)<sup>1</sup> is being finalized. [The WebSocket interface](http://www.w3.org/TR/websockets/#websocket)<sup>2</sup> provides, among others, the following:

- an attribute to define the connection URL to the server Endpoint (`url`)
- an attribute to know the status of the connection (`readyState` : CONNECTING, OPEN, CLOSING, CLOSED)
- some **Event Handler** in connection with the WebSocket lifecycle, eg :
  - the Event Handler `onopen` is called when a new connection is open
  - the Event Handler `onerror` is called when an error occurred during the communication
  - the Event Handler `onmessage` is called when a message arrives from the server
- methods (`send(DOMString data)`, `send(Blob data)`) with which it's possible to send different type of flow(text, binary) to the Endpoint server

## Example 4.1. Javascript source code example, from <http://websocket.org>

```
var wsUri = "ws://echo.websocket.org/";

function testWebSocket() {

    websocket = new WebSocket(wsUri);

    websocket.onopen = function(evt) { onOpen(evt) };
    websocket.onclose = function(evt) { onClose(evt) };
    websocket.onmessage = function(evt) { onMessage(evt) };
    websocket.onerror = function(evt) { onError(evt) }; }

}
```

---

<sup>1</sup> <http://w3.org/TR/websockets/>

<sup>2</sup> <http://www.w3.org/TR/websockets/#websocket>

## WebSocket

### Javascript API (Client)

---

```
function onOpen(evt) {
  writeToScreen("CONNECTED");
  doSend("WebSocket rocks");
}

function onClose(evt) {
  writeToScreen("DISCONNECTED");
}

function onMessage(evt) {
  writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data+'</span>');
  websocket.close();
}

function onError(evt) {
  writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
}

function doSend(message) {
  writeToScreen("SENT: " + message);
  websocket.send(message);
}
```

---

---

# Chapter 5. JSR 386 : Java API for WebSocket protocol

As the W3C defines how to use WebSocket in Javascript, the **Java Communittee Process (JCP)** does the same for the Java world via the JSR 386.

JSR 356 defines a [Java API for WebSocket protocol](http://jcp.org/en/jsr/detail?id=356)<sup>1</sup> which be part of **Java EE Web Profile** and give the ability to :

- create a **WebSocket Endpoint** (server or client), the name given to the Java component that can communicate via the WebSocket protocol
- the choice of **annotation** or programmatic approach
- **send and consume messages** controls, text or binary via this protocol
  - manage the message as a complete message or a sequence of partial messages
  - send or receive messages as Java objects (concept of **encoders / decoders**)
  - send messages **synchronously or asynchronously**
- configure and manage **WebSocket Session** (timeout, cookies...)



*The open source JSR-356 RI (Reference Implementation) is [the project Tyrus](https://tyrus.java.net/)<sup>2</sup>*

## 5.1. WebSocket Server Endpoint

The transformation of a Plain Old Java Object (POJO) to a **Server WebSocket Endpoint** (namely capable of handling requests from different customers on the same URI) is **very easy** since you only have to annotate the Java Class with **@ServerEndpoint** and one method with **@OnMessage** :

```
.....  
import javax.websocket.OnMessage;  
import javax.websocket.ServerEndpoint;  
  
@ServerEndpoint("/echo") ❶  
public class EchoServer {
```

---

<sup>1</sup> <http://jcp.org/en/jsr/detail?id=356>

<sup>2</sup> <https://tyrus.java.net/>

```
@OnMessage ❷  
public String handleMessage(String message) {  
    return "Thanks for the message: " + message;  
}  
  
}
```

---

- ❶ @ServerEndpoint transforms this POJO into a WebSocket Endpoint, the **value** attribute is mandatory in order to set the access URI to this Endpoint
- ❷ the *handleMessage* method will be invoked for each received message

## 5.2. Annotations

This Java API provides several types of annotations to be fully compatible with the WebSocket protocol :

Annotation	Role
@ServerEndpoint	Declare a Server Endpoint
@ClientEndpoint	Declare a Client Endpoint
@OnOpen	Declare this method handles open events
@OnMessage	Declare this method handles Websocket messages
@OnError	Declare this method handles error
@OnClose	Declare this method handles WebSocket close events

@ServerEndpoint attributes are listed below :

### value

relative URI or template URI (ex: "/echo", "/matches/{match-id}")

### decoders

list of message decoder classnames

### encoders

liste of message encoder classnames

### subprotocols

list of the names of the supported subprotocols (ex: <http://wamp.ws>)



## 5.3. Encoders and Decoders

As described earlier in this article, the Endpoint server can receive different types of content in messages : data in text format (JSON, XML ...) or binary format.

To effectively manage the messages from *peers client* or to them in the application business code, it is possible to create **Encoders and Decoders** Java classes.

Whatever the transformation algorithm, it will then be possible to transform :

- the business POJO to flow in the desired format for communication (JSON, XML, Binary ...)
- inflows in specific format(JSON, XML..) to the business POJO

Thus, the application code is structured so that the business logic is not affected by the type and format of messages exchanged between the *peer server* and *peers client* flows.

A concrete example is presented later in the article.

## 5.4. WebSocket Client Endpoint

This Java API also offers support for creating client-side Java Endpoints.

### Example 5.1. Java Client Endpoint sample

```
@ClientEndpoint
public class HelloClient {

    @OnMessage
    public String message(String message) {
        // code
    }
}

WebSocketContainer c = ContainerProvider.getWebSocketContainer();
c.connectToServer(HelloClient.class, "hello");
```

---

## Chapter 6. US OPEN Application

The sample application is deployed as a WAR outcome of a build with Apache Maven. In addition to the traditional management WebSocket lifecycle, the sending messages workflow is as follows :

- each *peer client* can connect to 1 or 4 live matches
- each *peer client* can disconnect from a match
- at each point of a match, clients which are connected to this match will receive datas (score, service...)
- the *peer client* may send a message to bet on the winner of the match
- each time that one *peer client* bet on a match, all others *peers clients* which have bet on the same match, will receive a message with the total number of bettors
- at the end of the match, *peers client* receive a message containing the name of the winner and a specific message if they bet on this match

**All messages are exchanged in JSON format**

The project layout is as follows :

### Example 6.1. Maven project structure

```
+ src/main/java
  |+ com.mgreau.wildfly.websocket
    |+ decoders
      |- MessageDecoder.java    ❶
    |+ encoders                 ❷
      |- BetMessageEncoder.java
      |- MatchMessageEncoder.java
    |+ messages                 ❸
      |- BetMessage.java
      |- MatchMessage.java
      |- Message.java
    |+ rest                     ❹
      |- RestApplication.java
      |- TournamentREST.java
      |- MatchEndpoint.java     ❺
      |- StarterService.java    ❻
      |- TennisMatch.java       ❼
  + src/main/resources
```

```

+ scr/main/webapp
  |+ css
  |+ images
  |+ js
    |+ matches ❸
      |- app.js
      |- controllers.js
      |- services.js
    |- websocket.js ❹
  |- index.html
  |- matches.html
pom.xml

```

- ❶ Decode JSON messages sent from the *peer client* (about bet on the winner) to a POJO (*BetMessage*)
- ❷ Encode in JSON format (via JSON-P), all messages about the winner and the match details for *peers clients*
- ❸ POJOs to handle messages sent between peers
- ❹ REST endpoint to list all tournament's matches
- ❺ The application WebSocket Server Endpoint (*peer server*)
- ❻ EJB `@Startup` in order to initialize this application at deployment time
- ❼ POJO to handle informations about the match
- ❽ AngularJS files to handle several matches (matches.html) with REST and WebSocket call
- ❾ File containing the implementation of Javascript API for WebSocket protocol to handle the client side of the communication for the simple case (index.html)

## 6.1. Maven dependencies for Java EE 7 API

### Example 6.2. pom.xml with Java EE 7 dependencies

```

<project>
...
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <!-- Java EE 7 -->
  <javaee.api.version>7.0</javaee.api.version>
</properties>

<dependencies>
  <dependency>

```

```

<groupId>javax</groupId> ❶
<artifactId>javaee-api</artifactId>
<version>${javaee.api.version}</version>
<scope>provided</scope>
</dependency>
</dependencies>
...
</project>

```

- ❶ It's important to use the Java EE 7 dependencies to be able to deploy the same application in multiple Java EE application servers (WildFly, Glassfish...) **without changing code**.

## 6.2. Add Server Endpoint

This endpoint can receive messages about betting on the winner of a match (identified by match-id) and it can also send to *peers client* all informations about the course of the match and bets.

### Example 6.3. Server Endpoint : MatchEndpoint.java

```

@ServerEndpoint(
    value = "/matches/{match-id}", ❶
    decoders = { MessageDecoder.class }, ❷
    encoders = { MatchMessageEncoder.class,
        BetMessageEncoder.class } ❸
)
public class MatchEndpoint {

    private static final Logger logger =
        Logger.getLogger("MatchEndpoint");

    /** All open WebSocket sessions */
    static Set<Session> peers = Collections.synchronizedSet(new
        HashSet<Session>());

    /** Handle number of bets by match */
    static Map<String, AtomicInteger> nbBetsByMatch = new
        ConcurrentHashMap<>();

    @Inject StarterService ejbService;

```

```
@OnOpen
public void openConnection(Session session,
    @PathParam("match-id") String matchId) { ❹
    logger.log(Level.INFO, "Session ID : " + session.getId() + " -
Connection opened for match : " + matchId);
    session.getUserProperties().put(matchId, true);
    peers.add(session);

    //Send live result for this match
    send(new MatchMessage(ejbService.getMatches().get(matchId)),
matchId);
}

public static void send(MatchMessage msg, String matchId) {
    try {
        /* Send updates to all open WebSocket sessions for this match */
        for (Session session : queue) {
            if
(Boolean.TRUE.equals(session.getUserProperties().get(matchId))) {
                if (session.isOpen()) {
                    session.getBasicRemote().sendObject(msg); ❺
                }
            }
        }
    } catch (IOException | EncodeException e) {
        logger.log(Level.INFO, e.toString());
    }
}

public static void sendBetMessage(Session session, BetMessage betMsg,
String matchId) {
    try {
        betMsg.setNbBets(nbBetsByMatch.get(matchId).get());
        session.getBasicRemote().sendObject(betMsg);
        logger.log(Level.INFO, "BetMsg Sent: {0}",
betMsg.toString());
    } catch (IOException | EncodeException e) {
        logger.log(Level.SEVERE, e.toString());
    }
}

@OnMessage
public void message(final Session session, BetMessage msg, ❻
    @PathParam("match-id") String matchId) {
    session.getUserProperties().put("bet", msg.getWinner());
}
```

```

//Send betMsg with bet count
if (!nbBetsByMatch.containsKey(matchId)) {
    nbBetsByMatch.put(matchId, new AtomicInteger());
}
nbBetsByMatch.get(matchId).incrementAndGet();
sendBetMessages(null, matchId, false);
}

@OnClose
public void closedConnection(Session session,
                               @PathParam("match-id") String
matchId) {
    if (session.getUserProperties().containsKey("bet")) {
        nbBetsByMatch.get(matchId).decrementAndGet();
        sendBetMessages(null, matchId, false);
    }
    /* Remove this connection from the queue */
    peers.remove(session);
}
...
}

```

- ❶ Access URI to this Endpoint, as the application context-root is *usopen*, the final URL looks like this : `ws://<host>:<port>/usopen/matches/1234`
- ❷ *MessageDecoder* transforms the incoming JSON flow (about the bet on the winner) into a POJO *BetMessage*
- ❸ This 2 encoders add the ability to transform from *MatchMessage* POJO and *BetMessage* POJO to messages in JSON format
- ❹ `@PathParam` annotation allows to extract part of the WebSocket request and pass the value (id match) as the parameter of the method, it is possible to manage several match with multiple clients for each match.
- ❺ Send, to connected peers, messages about the course of the match. Thanks to the *MatchMessageEncoder* object, simply pass the *MatchMessage* object.
- ❻ Handle received messages about the bet on the winner, thanks to the *MessageDecoder* object, one of the parameters of this method is a *BetMessage* object

## 6.3. Encodes and Decodes messages

To encode or decode messages exchanged between peers, simply implement the appropriate interface according to the message type (text, binary) and direction of processing (encoding, decoding), then redefine the associated method.

In the example below, it's the **encoder** for MatchMessage POJO to JSON format. The API used to perform this treatment is also a new API released with Java EE 7 : [Java API for JSON Processing \(JSON-P\)](http://jcp.org/en/jsr/detail?id=353)<sup>1</sup>

### Example 6.4. Text Encoder : MatchMessageEncoder.java

```
public class MatchMessageEncoder implements Encoder.Text<MatchMessage>
{

    @Override
    public String encode(MatchMessage m) throws EncodeException {
        StringWriter swriter = new StringWriter();
        try (JsonWriter jsonWrite = Json.createWriter(swriter)) {
            JsonObjectBuilder builder = Json.createObjectBuilder();
            builder.add(
                "match",
                Json.createObjectBuilder()
                    .add("serve", m.getMatch().getServe())
                    .add("title", m.getMatch().getTitle())
                    ...
            )

            jsonWrite.writeObject(builder.build());
        }
        return swriter.toString();
    }
}
```

## 6.4. HTML5 Web Client (single match - index.html)

The index.html page of this application loads the **websocket.js** file to implement the Javascript WebSocket API and thus interact with the Java Server Endpoint. This page handle only one single match.

---

<sup>1</sup> <http://jcp.org/en/jsr/detail?id=353>

## Example 6.5. API Javascript implemented into websocket.js

```

var wsUrl;

if (window.location.protocol == 'https:') { ❶
    wsUrl = 'wss://' + window.location.host + ':8443/usopen/
matches/1234';
} else {
    wsUrl = 'ws://' + window.location.host + ':8000/usopen/matches/1234';
}

function createWebSocket(host) {
    if (!window.WebSocket) { ❷
        ...
    } else {
        socket = new WebSocket(host); ❸
        socket.onopen = function() {
            document.getElementById("m1-status").innerHTML = 'CONNECTED...';
        };
        socket.onclose = function() {
            document.getElementById("m1-status").innerHTML = 'FINISHED';
        };
        ...
        socket.onmessage = function(msg) {
            try {
                console.log(data);
                var obj = JSON.parse(msg.data); ❹
                if (obj.hasOwnProperty("match")) { ❺
                    //title
                    m1title.innerHTML = obj.match.title;
                    // comments
                    m1comments.value = obj.match.comments;
                    // serve
                    if (obj.match.serve === "player1") {
                        m1p1serve.innerHTML = "S";
                        m1p2serve.innerHTML = "";
                    } else {
                        m1p1serve.innerHTML = "";
                        m1p2serve.innerHTML = "S";
                    }
                    ..
                }
                ...
            } catch (exception) {

```



```
data = msg.data;
console.log(data);
}
}
}
}
```

- ❶ Choose the appropriate WebSocket protocol according to the HTTP protocol currently used (secure or not)
- ❷ Check if the browser supports WebSocket API
- ❸ Create the WebSocket object
- ❹ Try to parse the JSON message sent by *peer server*, into the function called by `onmessage` Event Handler
- ❺ Check the received object type (MatchMessage or BetMessage) to achieve adequate treatment with DOM



To find out which browsers are compatible with **WebSocket API** visit the website [caniuse.com](http://caniuse.com)<sup>2</sup>. Today, the latest versions of browsers are compatible excepted for Android and Opera Mini Browser, which represent, both together, only 3% of web traffic.

## 6.5. AngularJS Client (several matches - matches.html)

As we saw on the beginning of this post, the version to handle severals matches is developed with AngularJS on the client side. So there are 3 JS files :

- app.js : just define the *tennisApp* angular application
- controllers.js : the TournamentCtrl controller
- services.js :
  - MatchWebSocketService : to handle WebSocket lifecycle with callback
  - TournamentRESTService : to get all matches from REST Endpoint server



This post is not a tutorial about AngularJS since it's my first sample with this framework. But it was a quick solution to handle several matches on the client side.

<sup>2</sup> <http://caniuse.com/#search=websocket>

## 6.6. Source code on Github

You can **fork this project on Github** at <https://github.com/mgreau/javaee7-websocket>

This sample application is basic, there could be many improvements like betting on other criteria...



'A feature that could be interesting technically, would be to create a new type of **bet based on the coordinates of each winning point**. Simply draw the ground through the HTML5 Canvas API and manage the coordinates selected by the user (such as winning point) and then compare with the actual coordinates at a point winner. '

## 6.7. Build and Deploy the WAR



Prerequisite :

- JDK 7
- Apache Maven 3.0.4+
- Java EE 7 Application Server : Wildfly 8 ou Glassfish 4

In order to build the WAR, you just have to execute the Maven command below ;

---

```
mvn clean package
```

---

If your application server is WildFly, you can quickly deploy the WAR with the command below (WildFly has to be started) :

---

```
mvn jboss-as:deploy
```

---

The usopen application is then available at :

- <http://localhost:8080/usopen/> for the simple case with only native Javascript
- <http://localhost:8080/usopen/matches> for the version with severals matches and AngularJS

---

# Chapter 7. Benchmark : WebSocket VS REST

In order to have some metrics about the performance of this new protocol, Arun Gupta has developed [an application that allows compare the execution time of<sup>1</sup>](https://github.com/arun-gupta/javaee7-samples/tree/master/websocket/websocket-vs-rest) the same treatment performed by WebSocket code and REST code.

Each endpoint (REST Endpoint and WebSocket Endpoint) just do an "echo" so they only return the flows they receive. The web interface of the application allows you to define the size of the message and the number of times that the message must be sent before the end of the test.

The benchmark results, shown below, are quite eloquent :

<b>Request</b>	<b>Total execution time REST Endpoint</b>	<b>Total execution time WebSocket Endpoint</b>
Sending 10 messages of 1 byte	220 ms	7 ms
Sending 100 messages of 10 bytes	986 ms	57 ms
Sending 1000 messages of 100 bytes	10 210 ms	179 ms
Sending 5000 messages of 1000 bytes	54 449 ms	1202 ms

---

<sup>1</sup> <https://github.com/arun-gupta/javaee7-samples/tree/master/websocket/websocket-vs-rest>

---

## Chapter 8. References about WebSocket

I would particularly recommend [Arun Gupta](#)<sup>1</sup>'s conferences, which allow you in less than 1 hour to discover and understand the WebSocket technology in general and the Java API for WebSocket.

For more advanced information, the ideal is IETF, W3C and Java specifications.

[RFC 6455: The WebSocket Protocol](#)<sup>2</sup> - *IETF Specification*

[W3C: The WebSocket API](#)<sup>3</sup> - *W3C Specification* (Candidate Recommendation)

[JSR 356: Java API for WebSocket Protocol](#)<sup>4</sup> - *Java Specification*

[Adopt a JSR - JSR 356](#)<sup>5</sup>

[Java EE 7 & WebSocket API](#)<sup>6</sup> - *Arun Gupta's conference @ SF* (from the 46th minute)

[Getting Started with WebSocket and SSE](#)<sup>7</sup> - *Arun Gupta's conference @ Devovx UK 2013*

*This article was structured based on the UK 2013 Devovx conference.*

---

<sup>1</sup> <https://twitter.com/arungupta>

<sup>2</sup> <http://tools.ietf.org/html/rfc6455>

<sup>3</sup> <http://w3.org/TR/websockets/>

<sup>4</sup> <http://jcp.org/en/jsr/detail?id=356>

<sup>5</sup> <https://glassfish.java.net/adoptajsr/jsr356.html>

<sup>6</sup> <http://www.youtube.com/watch?v=QqbuDFIT5To>

<sup>7</sup> <http://www.parleys.com/play/51c1ccea4b0ed8770356828/chapter4/about>

---

## Chapter 9. Conclusion

This article has introduced, through a concrete example, **the WebSocket protocol, the HTML5 WebSocket API and Java API for WebSocket released with Java EE 7**. It was already possible to use WebSocket with Java frameworks like [Atmosphere](#)<sup>1</sup> but lacked a standard.

Today all **standards are completed or about to be**, this new technology meets a specific need and is promising in terms of performance. To be heavily used, this protocol will need to be allowed in businesses where often only the HTTP protocol is permitted.

---

<sup>1</sup> <http://async-io.org/download.html>