Java EE 7 et l'API Java pour WebSocket (JSR 356)

Maxime Gréau - http://mgreau.com

Table of Contents

List of Figures

List of Examples

Cet article présente et met en oeuvre à travers un exemple concret et disponible en ligne[1], une des 4 nouvelles JSRs de **Java EE 7**[2], à savoir **l'API Java pour communiquer via le protocole WebSocket (JSR 356)**[3], .

Après la lecture de cet article, vous devriez être en mesure de comprendre la définition de ce qu'il est possible de faire avec le protocole WebSocket, donnée lors de Devoox UK par Arun Gupta[4] :

> WebSocket gives you bidirectionnal, full duplex, communication channel over a single TCP.
>
> — Arun Gupta (Java EE Evangelist chez Oracle) - Devoxx UK 2013

1 http://wildfly-mgreau.rhcloud.com/usopen/
2 http://jcp.org/en/jsr/detail?id=342
3 http://jcp.org/en/jsr/detail?id=356
4 https://twitter.com/arungupta

Chapter 1. Introduction à Java EE 7

La **Plateforme Java Entreprise Edition** est sortie en version 7 (Java EE 7) au mois de Juin 2013. Dans la continuité des versions Java EE 5 et Java EE 6, **Java EE 7** propose toujours de simplifier le travail du développeur. Cette version agrémente les versions précédentes avec 3 objectifs principaux :

- s'interfacer avec **HTML5** (WebSocket API, JSON-P API, JAX-RS)
- avoir une **meilleure productivité** (JMS API)
- répondre aux **besoins des entreprises** (Batch API)

Figure 1.1. Les 3 objectifs de Java EE 7

Java Platform, Entreprise Edition 7 (JSR 342), se résume donc autour de :

- 4 nouvelles spcécifications : `Java API for WebSocket 1.0`, `Java API for JSON Processing 1.0`, `Batch Applications 1.0`et `Concurrency Utilities for Java EE 1.0`
- 3 spécifications avec une mise à jour majeure : `JMS 2.0`, `JAX-RS 2.0`et `EL 3.0`
- ainsi que 7 spécifications mises à jour dans une version mineure : `JPA 2.1`, `Servlet 3.1`, `EJB 3.2`, `CDI 1.1`, `JSF 2.2` et `Bean Validation 1.1`

Chapter 2. DEMO : Application HTML5 / JSR-356 API déployée sur Wildfly 8 (OpenShift)

Les plus impatients peuvent accéder à la `démonstration en ligne`[1] du code qui va être, en partie, expliqué dans cet article. Il s'agit d'une application qui permet :

- de suivre un match de Tennis en Live (Finale de l'US Open 20013) sans aucune action autre que la connexion à l'URL
- de parier sur le vainqueur du match

Vous allez me dire : "Rien d'extraordinaire !", et vous aurez raison.

A première vue, ce sont des choses que nous connaissons déjà sur beaucoup d'applications aujourd'hui, mais celle-ci est intéressante techniquement car comme vous le verrez au cours de l'article, tout est basé sur du **standard autour du nouveau protocole WebSocket (ws:// ou wss://)** et non sur du "hacking" de protocole HTTP.

Les technologies utilisées pour le developpement de cette application sont :

> - côté client : `HTML5`, `CSS`, `Javascript (WebSocket API)` avec *Bootstrap CSS mais sans JQuery ou BootstrapJS*
> - côte serveur : `Java API for WebSocket`, `EJB`, `JSON-P`

[1] http://wildfly-mgreau.rhcloud.com/usopen/

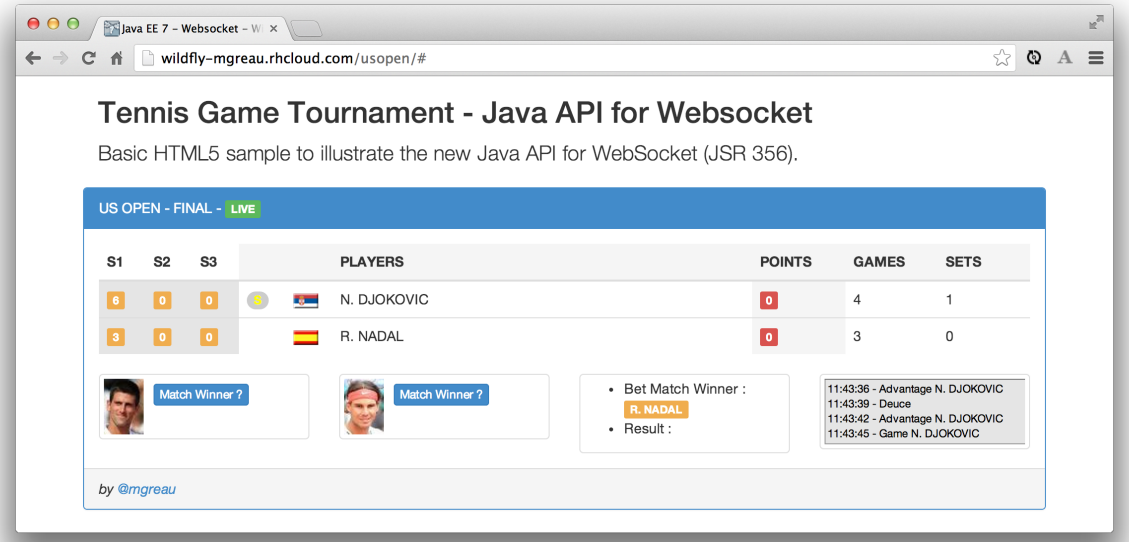

Figure 2.1. Mise en oeuvre des WebSocket (Java API et Javascript API)

Non cette démonstration n'est **pas une application de chat :)** Il est évident que la démo "chat" est celle qui vient en premier à l'esprit pour illustrer l'utilisation de la technologie WebSocket. Néanmoins, il existe beaucoup d'autres cas d'utilisation, comme par exemple le travail collaboratif sur un document texte en ligne. Ou encore les jeux en ligne comme le jeu d'échec présenté lors de la keynote de JavaOne 2013[2].

Cette application est disponible sur le Cloud grâce à OpenShift[3], la solution Cloud de RedHat. Elle est déployée sur le serveur d'applications Wildfly 8.0.0-Beta3 (normalement certifié Java EE 7 fin 2013). Pour mettre en place un serveur de ce type, il suffit de suivre le post de Shekhar Gulati[4]

2 https://blogs.oracle.com/javaone/entry/the_javaone_2013_technical_keynote
3 https://www.openshift.com/
4 https://www.openshift.com/blogs/deploy-websocket-web-applications-with-jboss-wildfly

Chapter 3. WebSocket (WS) : un nouveau protocole différent de HTTP

HTTP[1] est le protocole standard utilisé pour le Web, il est très efficace pour certains cas d'utilisation mais il dispose néanmoins de **quelques inconvénients** dans le cas **d'applications Web intéractives** :

- **half-duplex** : basé sur le pattern request/response, le client envoi une requête puis le serveur réalise un traitement avant de renvoyer une réponse, le client est donc contraint d'attendre une réponse du serveur
- **verbose** : beaucoup d'informations sont présentes avec les headers HTTP associés au message, aussi bien dans la requête HTTP que dans la réponse HTTP
- pour faire du **server push**, il est nécessaire d'utiliser des méthodes de contournement (polling, long polling, Comet/Ajax) car il n'existe pas de standard.

Ce protocole n'est donc pas optimisé pour scaler sur des applications qui ont d'important besoins de communication temps réel bi-directionnelle. C'est pourquoi le **nouveau protocole WebSocket** propose des fonctionnalités plus évoluées que HTTP, puisqu'il est :

- basé sur `1 unique connexion TCP entre 2 peers`(en HTTP chaque requête/réponse necessite une nouvelle connexion TCP)
- `bi-directionnel`: le client peut envoyer un message au serveur et le serveur peut envoyer un message au client
- `full-duplex`: le client peut envoyer plusieurs messages vers le serveur et le serveur vers le client sans attendre de réponse l'un de l'autre

> *Le terme **client** est utilisé uniquement pour définir celui qui va initialiser la connexion. Dès lors que la connexion est établie, le client et le serveur deviennent tous les deux des `peers` avec les mêmes pouvoirs l'un par rapport à l'autre.*

Le protocole WebSocket devait à l'origine faire partie de la spécification HTML5 mais comme celle-ci sortira officiellement en 2014, il est finalement défini, au même titre que HTTP, par une spécification IETF, la RFC 6455[2].

[1] http://tools.ietf.org/html/rfc2616
[2] http://tools.ietf.org/html/rfc6455

Comme le montre le schéma ci-après, le **protocole WebSocket fonctionne en 2 phases** nommées :

1. `handshake`
2. `data transfer`

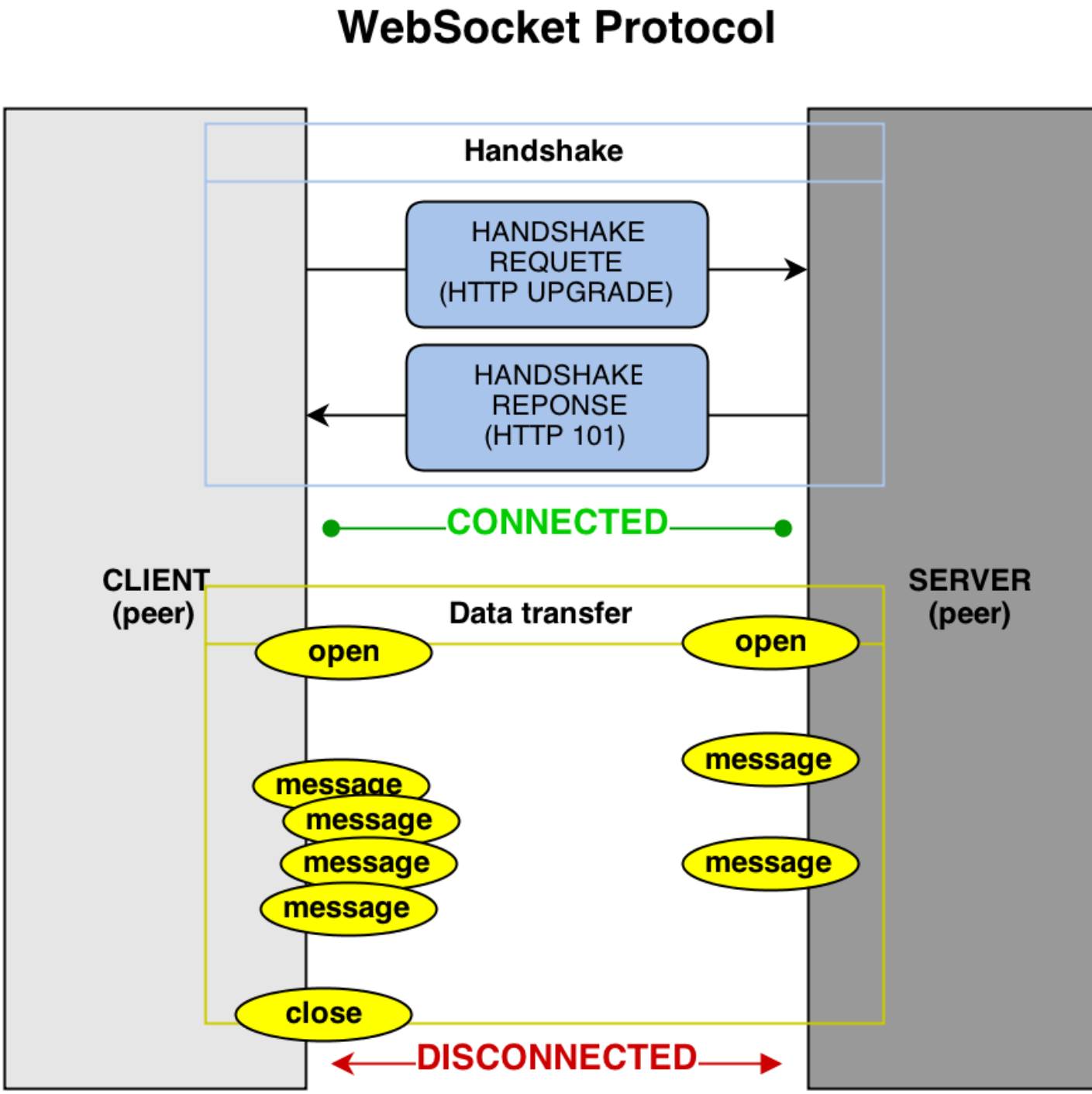

Figure 3.1. Explication du protocole WebSocket

3.1. Handshake

La phase nommée **Handshake** correspond à un **unique échange requête/réponse HTTP** entre l'initiateur de la connexion (peer client) et le peer serveur. Cet échange HTTP est spécifique car il utilise la notion **d'Upgrade, définie dans la spécification HTTP.**[3]

Le principe est simple : **l'Upgrade HTTP** permet au client de communiquer avec le serveur pour lui demander de changer de protocole de communication et ainsi faire en sorte que le client et le serveur utilisent un protocole autre que HTTP pour discuter.

Example 3.1. Exemple de Requête HTTP Handshake

```
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 Methode HTTP GET et version 1.1 obligatoires
❷ Host utilisé pour la connexion WebSocket
❸ Demande d'Upgrade vers le protocole WebSocket
❹ Demande d'Upgrade HTTP pour changer de protocole

Example 3.2. Exemple de Réponse HTTP Handshake

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

❶ Code HTTP 101, le serveur est compatible et accepte le changement de protocole
❷ L'upgrade vers le protocole WebSocket est accepté

[3] http://tools.ietf.org/html/rfc2616#section-14.42

Lorsque la demande d'upgrade du protocole HTTP vers le protocole Web Socket a été validée par le serveur endpoint, il n'y a plus de communication possible en HTTP, tous les échanges sont réalisés via le protocole WebSocket.

3.2. Data transfer

Une fois que le **handshake** est acceptée, la mise en place du protocole WebSocket est donc acquise. Une connexion côté *peer server* est ouverte ainsi que côté *peer client*, une gestion de callback est activée pour initier la communication.
La phase de **Data transfer** peut alors entrer en jeu, c'est-à-dire que les 2 peers peuvent désormais **s'échanger des messages dans une communication bi-directionnelle et full-duplex**.

Comme le montre le schéma de la **Figure 3**, le `peer server` peut envoyer plusieurs messages (dans l'exemple : 1 message à chaque point du match) sans aucune réponse du `peer client` qui, lui, peut également envoyer des messages à n'importe quel moment (dans l'exemple : le pari sur le vainqueur du match). Chaque peer peut envoyer un message spécifique afin de clôturer la connexion.

Dans Java EE7, le code côté `peer server` est en **Java** alors que le code côté `peer client` est en **Java ou en Javascript**.

Chapter 4. WebSocket Javascript API (Client)

Pour communiquer à partir d'une application Web avec un serveur en utilisant le protocole WebSocket, il est nécessaire d'utiliser **une API cliente en Javascript**. C'est le W3C qui définit cette API.
La spécification W3C de cette API Javascript pour WebSocket[1] est en cours de finalisation. L'interface WebSocket[2] propose, entre-autres, les éléments suivants :

- un attribut pour l'URL de connexion au server Endpoint (`url`)
- un attribut sur l'état de la connexion (`readyState` : CONNECTING, OPEN, CLOSING, CLOSED)
- des **Event-Handler (gestionnaire d'évènement)** pour s'adapter aux méthodes du cycle de vie des WebSocket, par exemple :
  - l'Event-Handler `onopen` est appelé lorsqu'une nouvelle connexion est initiée
  - l'Event-Handler `onerror` est appelé lorsqu'une erreur est reçue pendant la communication
  - l'Event-Handler `onmessage` est appelé lorsqu'un message est reçu
- les méthodes (`send(DOMString data)` , `send(Blob data)`) avec lesquelles il est possible d'envoyer différents types de flux (texte, binaire) vers le serveur Endpoint

Example 4.1. Exemple de code Javascript, issue de 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) };
```

[1] http://w3.org/TR/websockets/
[2] http://www.w3.org/TR/websockets/#websocket

```
 websocket.onerror = function(evt) { onError(evt) }; }
}

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 pour WebSocket

Le W3C définit donc comment utiliser WebSocket en Javascript, le **Java Communitee Process (JCP)** fait de même pour le monde Java via la JSR 386 .
La JSR 356 définit ainsi une API Java pour WebSocket[1] qui propose :

- la création d'un `WebSocket Endpoint` (serveur ou client), nom donné au composant Java capable de communiquer via le protocole WebSocket
- la possibilité d'utiliser l'approche par **annotation Java** ou par programmation
- la possibilité **d'envoyer et de consommer des messages** de contrôles, textuels ou binaires via ce protocole
  - de gérer le message en tant que message complet ou par une séquence de messages partiels
  - envoyer ou recevoir les messages en tant qu'objets Java (notion d'**encoders/decoders**)
  - envoyer les messages **en synchrone ou en asynchrone**
- la configuration et la **gestion des sessions WebSocket** (timeout, cookies…)
- une intégration dans **Java EE Web Profile**

L'implémentation de référence Java pour l'API WebSocket est le projet Tyrus[2]

5.1. WebSocket Endpoint : Serveur

La transformation d'un Plain Old Java Object (POJO) vers un **WebSocket Endpoint** de type serveur (c'est-à-dire capable de gérer des requêtes de plusieurs clients sur une même URI) est **extrêment simple**, puisqu'il suffit d'annoter la classe avec **@ServerEndpoint** et une méthode du POJO avec **@OnMessage** :

```
import javax.websocket.OnMessage;
import javax.websocket.ServerEndpoint;
```

[1] http://jcp.org/en/jsr/detail?id=356
[2] https://tyrus.java.net/

```
@ServerEndpoint("/echo") ❶
public class EchoServer {

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

}
```

❶ L'annotation @ServerEndpoint transforme le POJO en WebSocket Endpoint, l'attribut **value** est obligatoire afin de préciser l'URI d'accès à cet Endpoint

❷ la méthode handleMessage sera évoquée lors de chaque message reçu

5.2. Annotations

L'API met à disposition plusieurs types d'annotations afin d'être entièrement compatible avec le procotole WebSocket :

Annotation	Rôle
@ServerEndpoint	Déclare un Server Endpoint
@ClientEndpoint	Déclare un Client Endpoint
@OnOpen	Défini la méthode appelée pour gérer l'évenement d'ouverture de la connexion
@OnMessage	Défini la méthode appelée pour gérer l'évenement de réception d'un message
@OnError	Défini la méthode appelée pour gérer l'évenement lors d'une erreur
@OnClose	Défini la méthode appelée pour gérer l'évenement de clôture de la connexion

Les attributs de l'annotation `@ServerEndpoint` sont les suivants :

value
: URI relative ou URI template (ex: "/echo", "/chat/{subscriver-level}")

decoders
: liste de noms de classes utilisées pour décoder les messages entrants

encoders
: liste de noms de classes utilisées pour encoder les messages sortants

subprotocols
liste de sous-protocoles autorisés (ex: http://wamp.ws)

5.3. Encoders et Decoders

Comme il a été décrit plus tôt dans cet article, le serveur Endpoint peut recevoir différents types de contenu dans les messages : des données au format texte (JSON, XML…) ou au format binaire.
Afin de gérer efficacement les messages provenant des *peers client* ou à destination de ceux-ci dans le code métier de l'application, il est possible de créer des classes Java de type **Decoders et Encoders**.

Quelque soit l'algorithme de transformation, il va alors être possible de transformer :

- le POJO métier vers un flux au format désiré pour l'envoi (JSON, XML, Binaire…)
- les flux entrants dans format spécfique (JSON, XML..) vers le POJO métier

Ainsi, le code de l'application est organisé de telle façon que la logique métier n'est pas impactée par le type et le format de flux échangés entre le *peer serveur* et les *peers clients*. Un exemple concret est présenté dans la suite de l'article.

5.4. WebSocket Endpoint : Client

L'API propose donc également le support pour créer des Endpoints côté client en Java.

Example 5.1. Exemple de Client Endpoint en Java

```
@ClientEndpoint
public class HelloClient {

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

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

Chapter 6. Application US OPEN

L'application exemple est déployée sous forme de WAR issue d'un projet Apache Maven. Outre la gestion classique du cycle de vie WebSocket, le workflow d'envoi de messages est le suivant :

- à chaque point du match, les *peers clients* recoivent les données du match (score, service…)
- le *peer client* peut envoyer un message pour parier sur le gagnant du match
- à la fin du match, les *peers clients* reçoivent un message contenant le nom du vainqueur

Tous les messages sont échangés au format JSON.

L'arborescence du projet est la suivante :

Example 6.1. Structure du projet Maven

```
+ 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
      |- MatchEndpoint.java        ❹
      |- StarterService.java       ❺
      |- TennisMatch.java          ❻
+ src/main/resources
+ scr/main/webapp
   |+ css
   |+ images
   |- index.html
   |- websocket.js                ❼
pom.xml
```

❶ Decode le message JSON provenant du *peer client* concernant le pari sur le vainqueur en POJO (*BetMessage*)
❷ Encode à destination des *peers clients*, en JSON (via JSON-P), les messages contenant le détail du match et le résultat du pari sur le vainqueur
❸ POJOs représentant les types de messages échangés entre peers
❹ WebSocket Server Endpoint de l'application (*peer server*)
❺ EJB @Startup permettant d'initialiser l'application lors du déploiement
❻ POJO pour gérer les informations du match
❼ Fichier Javascript pour la communication WebSocket du *peer client* via l'API Javascript

6.1. Dépendances Maven Java EE 7

Example 6.2. pom.xml avec les dépendances Java EE 7

```
<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 il est important d'utiliser les dépendances de la spécification Java EE 7 afin de pouvoir déployer l'application dans plusieurs serveurs d'applications Java EE sans changement de code (Wildfly, Glassfish…)

6.2. Créer le Server Endpoint

Cet Endpoint permet de recevoir les messages concernant les paris sur le vainqueur du match et également d'envoyer aux *peers clients* les informations du déroulement du match.

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");

 /* Queue for all open WebSocket sessions */
 static Queue<Session> queue = new ConcurrentLinkedQueue<>();

 @OnOpen
 public void openConnection(Session session,
    @PathParam("match-id") String matchId) { ❹
     /* Register this connection in the queue */
     queue.add(session);
     session.getUserProperties().put(matchId, true);
     logger.log(Level.INFO, "Connection opened for game : " +
 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
        logger.log(Level.INFO, "Score Sent: {0}", msg);
         }
       }
```

```
        }
      } catch (IOException | EncodeException e) {
        logger.log(Level.INFO, e.toString());
      }
    }

    @OnMessage
    public void message(final Session session, BetMessage msg) { ❻
        logger.log(Level.INFO, "Received: Bet Match Winner - {0}",
    msg.getWinner());
        session.getUserProperties().put("betMatchWinner", msg);
    }
...
}
```

❶ URI pour accéder à cet Endpoint, comme le context-root de l'application est */usopen*, un exemple d'URL est `ws://<host>:<port>/usopen/matches/1234`

❷ la classe *MessageDecoder* permet de transformer le flux JSON entrant pour le pari sur le vainqueur en POJO *BetMessage*

❸ les 2 encodeurs permettent de transformer les POJO *MatchMessage* et *BetMessage* en flux JSON

❹ l'annotation `@PathParam` permet ici d'extraire l'élément de la requête WS et de passer la valeur (identifiant du match) en paramètre de la méthode, il est ainsi possible de gérer plusieurs matchs avec des clients différents pour chaque match.

❺ Envoi du message concernant le match aux peers connectés, grâce à l'*Encoder* il suffit de passer en paramètre un objet *MatchMessage*

❻ Gestion de la réception des messages de pari sur le vainqueur du macth, grâce au *Decoder* la méthode prend en paramètre un objet *BetMessage*

6.3. Encoder et Décoder les messages échangés

Pour encoder ou décoder les messages échangés entre peers, il suffit d'implémenter l'interface adéquate selon le type de message (Texte, Binaire) et le sens de traitement (encodage, décodage), puis de redéfinir la méthode associée.
Dans l'exemple ci-dessous, il s'agit de l'encodeur pour le POJO MatchMessage vers le format JSON. L'API utilisée pour réaliser ce traitement est une des nouvelles API de Java EE 7 : Java API for JSON Processiong (JSON-P)[1]

[1] http://jcp.org/en/jsr/detail?id=353

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. Client Web HTML5

L'unique page HTML de cette application charge le fichier **websocket.js** pour mettre en oeuvre l'API Javascript WebSocket et ainsi intéragir avec le Server Endpoint Java.

Example 6.5. API Javascript : 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 Choix du protocole WS selon le type de protocole HTTP utilisé (sécurisé ou non)

❷ Test du support par le navigateur de l'API WebSocket

❸ Création du WebSocket

❹ Sur l'Event-Handler `onmessage` , traitement du flux JSON reçu via le *peer serveur*

❺ Test du type d'objet reçu (Match ou Pari) afin de réaliser le traitement adéquat avec le DOM

Pour savoir quels sont les **navigateurs compatibles avec l'API WebSocket**, consultez le site caniuse.com[2]. Aujourd'hui, les dernières versions des navigateurs sont compatbiles exceptées pour Opéra mini et Android Browser, qui représentent, à eux deux, seulement 3% du traffic web.

6.5. Sources de l'exemple sur Github

Vous pouvez **forker le code sur Github** à l'URL `https://github.com/mgreau/javaee7-websocket`

Cette application exemple est très basique, les idées d'améliorations possibles sont nombreuses : gérer un tournoi avec plusieurs matchs, parier sur d'autres critères, voir en live les paris des autres internautes…

*Une feature, qui serait particulièrement intéressante techniquement, serait de créer un nouveau type de pari sur **la zone de terrain des points gagnants**. Il suffit de dessiner le terrain grâce à l'API HTML5 Canvas et de gérer les coordonnées de l'emplacement cliqué par l'internaute (comme zone gagnante) puis de les comparer aux coordonnées réelles lors d'un point gagnant.*

6.6. Construire et Déployer le WAR

Pré-requis :

- JDK 7
- Apache Maven 3.0.4+
- Serveur d'applications Java EE 7 : Wildfly 8 ou Glassfish 4

Pour créer l'archive WAR, il suffit d'executer la commande Apache Maven ci-dessous ;

```
mvn clean package
```

Si vous utilisez Wildfly, le déploiement est automatique (le serveur doit être démarré) avec la commande ci-dessous :

[2] http://caniuse.com/#search=websocket

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

Il suffit ensuite d’accéder à l’URL : http://localhost:8080/usopen/

Chapter 7. Performances : WebSocket vs REST

Afin d'avoir des métriques concernant les performances de ce nouveau protocole, Arun Gupta a developpé une application qui permet de comparer les temps d'execution[1] d'un même traitement réalisé avec du code développé en utilisant les technologies WebSocket et REST.

Les 2 endpoints de l'application (REST Endpoint et WebSocket Endpoint) ne font que renvoyer le flux qu'ils recoivent. L'interface Web de cette application permet de définir la taille du message et le nombre de fois que ce message doit être envoyé avant la fin du test.

Les résultats de ses tests, présentés ci-dessous, sont éloquents :

Type de Requête	**Temps execution REST Endpoint**	**Temps execution WebSocket Endpoint**
Envoi de 10 messages de 1 byte	220 ms	7 ms
Envoi de 100 messages de 10 bytes	986 ms	57 ms
Envoi de 1000 messages de 100 bytes	10 210 ms	179 ms
Envoi de 5000 messages de 1000 bytes	54 449 ms	1202 ms

[1] https://github.com/arun-gupta/javaee7-samples/tree/master/websocket/websocket-vs-rest

Chapter 8. Références pour tout savoir sur les WebSocket

Je vous recommande plus particulièrement les conférences d' Arun Gupta[1], qui vous permettent, en moins d'1 heure, de tout connaître/comprendre sur la technologie WebSocket en général et sur l'API Java en particulier.
Pour des informations plus avancées, l'idéal reste les spécifications IETF, W3C et Java.

RFC 6455: The WebSocket Protocol[2] - *Spécification IETF*

W3C: The WebSocket API[3] - *Spécification W3C* (Candidate Recommandation)

JSR 356: Java API for WebSocket Protocol[4] - *Spécification Java*

Adopt a JSR - JSR 356[5]

Java EE 7 & WebSocket API[6] - *Conférence Arun Gupta SF* (à partir de la 46e minute)

Getting Started with WebSocket and SSE[7] - *Conférence Arun Gupta Devoxx UK 2013*

Cet article a été structuré en se basant sur la conférence Devoxx UK 2013.

1 https://twitter.com/arungupta
2 http://tools.ietf.org/html/rfc6455
3 http://w3.org/TR/websockets/
4 http://jcp.org/en/jsr/detail?id=356
5 https://glassfish.java.net/adoptajsr/jsr356.html
6 http://www.youtube.com/watch?v=QqbuDFIT5To
7 http://www.parleys.com/play/51c1cceae4b0ed8770356828/chapter4/about

Chapter 9. Conclusion

Cet article a introduit, grâce à un exemple concret, **le protocole WebSocket, l'API WebSocket HTML5 et l'API Java pour les WebSocket sortie avec Java EE 7**. Il était déjà possible d'utiliser les WebSocket en Java grâce à des frameworks comme Atmosphere mais il manquait un standard.
Aujourd'hui tous **les standards sont finalisés ou en passe de l'être**, cette nouvelle technologie répond à un besoin précis et est prometteuse en terme de performance. Pour qu'elle soit massivement utilisée, il faudra tout de même que ce protocole soit autorisée dans les entreprises là où bien souvent seul le protocole HTTP est disponible.