Introduction aux API WebSocket

R4.01 - Architecture Logicielle

Luc Klaine

Table des matières

WebSocket

Qu’est-ce que c’est ?

- **Officiellement** :
  - **WebSocket (WS)** est un standard du web désignant un protocole réseau de la couche application et une interface de programmation visant à créer des canaux de communication full-duplex par-dessus une **connexion TCP** pour les navigateurs web.
- **Concrètement** :
  - Il s’agit d’une nouvelle fonctionnalité de **HTML5** qui permet de diffuser des données vers et depuis un navigateur web.

WebSocket

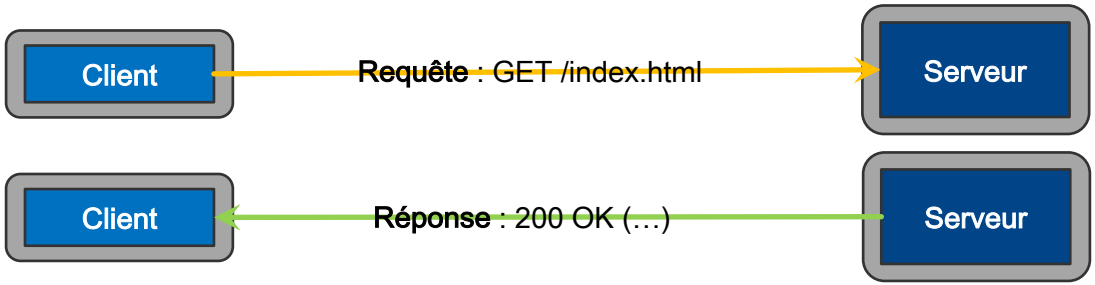
Quelques caractéristiques techniques

- Définit **un nouveau protocole** : ‘ws://’ et ‘wss://’
  - Normalisé par l'IETF dans la RFC 64552 en 2011 et l'interface de programmation par le W3C.
  - Commence comme HTTP et bascule ensuite sur les spécificités WS.
- Utilise les ports classiques : 80, 8080 et 443
  - 443 est recommandé en cas de passage par un PROXY.
- **Supporte les connexions multiples sur une seule socket TCP.**

WebSocket

Comparaison HTTP / WS

- **Connexion HTTP :**
  - Le client envoie une requête au serveur qui lui répond.



- **Connexion WS :**
  - Les données sont envoyées dans les deux sens à tout moment.

WebSocket

Comparaison HTTP / WS

- **Moins de temps de latence** :
  - Pas de nouvelle connexion **TCP** à chaque requête **HTTP**.
- **Moins de ressources** :
  - Chaque message envoie quelques octets au lieu de l'entête complet **HTTP**.
- **Moins de trafic** :
  - Les clients n'ont pas besoin d'envoyer des requêtes régulières pour écouter les messages qui ne sont envoyés que lorsqu'il y a des données.

- **Nécessité de surveiller la connexion et de se reconnecter,**
- **Nécessité de mettre en tampon et de renvoyer les messages :**
  - Lors d'une rupture réseau ou d'un 'timeout'.

WebSocket

Quelques cas d'usages

- **Pour les applications 'temps réel'** :
  - Lorsqu'une faible latence entre la requête et la réponse sont nécessaires. (Gaming, Collaboratif, Tableau de bord, Tracking, Chat, etc.)
- **Pour éviter les 'bidouillages'** :
  - Comme des requêtes régulières diffusées par exemple via un `<iframe>` caché.
  - **Lents, souvent complexes et volumineux**.
- **Pour éviter l'installation de 'plugins'** :
  - Qui ne fonctionnent pas partout.

Table des matières

Côté Client

API Javascript

- **Initialisation** :
  - `const socket = new WebSocket('ws://example.com')`
- **Envoi de messages** :
  - `send(_data)` : où `_data` peut être du texte ou une donnée binaire
- **Événements en provenance du serveur** :
  - onopen(_event) : lorsque la connexion avec le serveur est ouverte.
  - onclose(_event) : lorsque le serveur a fermé la connexion.
  - onerror(_event) : lorsqu'il y a une erreur de connexion.
  - onmessage(_event) : lors de la réception d'un message où `_event.data` est le contenu du message qui peut être du texte ou une donnée binaire.

Table des matières

Côté Serveur

API Jakarta EE

- L'encapsulation de la socket de l'**API WebSocket** au sein du **Framework Jakarta EE** ne repose pas sur une classe ou une interface particulière afin de pouvoir modifier les paramètres de l'adresse de connexion et ceux des méthodes de gestion des événements à l'envie.
- N'importe quelle classe utilisateur peut donc encapsuler la socket, elle et ses méthodes doivent simplement être préalablement **annotées**.
- Les événements côté serveur sont symétriques par rapport à ceux côté client. Ils ont les mêmes fonctions et ils ont les mêmes noms.

Côté Serveur

API Jakarta EE

- Code de référence pour créer une socket :

```
@ServerEndpoint(value = "/WebSocket/{username}", configurator = MyWebSocket.EndpointConfigurator.class)
public class MyWebSocket {
     /** Singleton. */
     private static MyWebSocket s_instance = new MyWebSocket();
     public static MyWebSocket getInstance() {
          return s_instance;
     }

     /** Permet de ne pas avoir une instance différente par client. */
     public static class EndpointConfigurator extends ServerEndpointConfig.Configurator {
          @Override
          @SuppressWarnings("unchecked")
          public <T> T getEndpointInstance(Class<T> _endpointClass) {
               return (T) s_instance ;
          }
     }
}
```

- `@ServerEndPoint` est l'annotation de la classe.
- `ws://localhost:8080/WebSocket/{username}` est l'adresse de connexion où `{username}` désigne n'importe quel nom d'utilisateur qui pourra être récupéré lors de l'événement d'ouverture de la connexion.

Côté Serveur

API Jakarta EE

- Code de référence pour ouvrir une connexion :

```java
/**
 *   Réagit à l'ouverture de la connexion.
 *   @param _session La session.
 *   @param _username Le nom de l'utilisateur.
 */
@OnOpen
public void onOpen(Session _session, @PathParam("username") String _username) {
}
```

- `@OnOpen` est l'annotation de la méthode d'ouverture de la connexion.
- La méthode s'appelle `onOpen()` mais pourrait s'appeler n'importe comment.
- La **session courante** `_session` est récupérée comme premier paramètre.
- Le **paramètre de l'adresse de connexion** `_username` correspondant à `{username}` est récupéré comme second paramètre préalablement annoté par `@PathParam("username")`

Côté Serveur

API Jakarta EE

- Code de référence pour fermer une connexion :

```java
/**
 *   Réagit à la fermeture de la connexion.
 *   @param _session La session.
 *   @param _reason La raison de la fermeture.
 */
@OnOpen
public void onClose(Session _session, CloseReason _reason) {
}
```

- `@OnClose` est l'annotation de la méthode de fermeture de la connexion.
- La méthode s'appelle `onClose()` mais ce n'est toujours pas obligatoire.
- La **session courante** `_session` est récupérée comme premier paramètre.
- La **raison de la fermeture** `_reason` est récupérée comme second paramètre.

Côté Serveur

API Jakarta EE

- Code de référence pour recevoir un message :

```java
/**
 *   Réagit à un message.
 *   @param _content Le contenu.
 *   @param _session La raison de la fermeture.
 */
@OnMessage(maxMessageSize = 32768)
public void onMessage(String _content, Session _session) {
}
```

- `@OnMessage` est l'annotation de la méthode de réception de message, on peut fixer la taille maximum du message, ici `maxMessageSize = 32768`.
- La méthode s'appelle `onMessage()` mais ce n'est toujours pas obligatoire.
- Le **contenu du message** est récupéré comme premier paramètre.
- La **session courante** est récupérée comme second paramètre.

Côté Serveur

API Jakarta EE

- Une **session** est passée en paramètre à chaque méthode : normal car une seule socket peut gérer de multiples connexions identifiées par leurs sessions. La trace de ces sessions doit être conservées dans une `map` dans la socket :

```
/** La collection de sessions. */
private Hashtable<String, Session> m_sessions = new Hashtable<>() ;
```

  - Où la clé est la valeur unique `_session.getId()` et la valeur `_session`.
  - Chaque session a une méthode `_session.getProperties()` qui contient des propriétés utilisateurs qui peut être modifiées et consultées à l'envie.

- Chaque **session** peut ainsi recevoir un message :

```
try (Writer writer = _session.getBasicRemote().getSendWriter()) {
     writer.write(_text);
}
```

  - Où `_text` est simplement une chaine de caractère.

Introduction aux API WebSocket

R4.01 - Architecture Logicielle

Luc Klaine

Avez-vous des questions ?

luc.klaine@univ-rennes.fr