WebSocket

🏁 29:00

Il protocollo **RFC6455** è un protocollo molto diffuso per particolari tipi di interazione tra un browser e server web; nasce con l'obbiettivo di superare i limiti dovuti alla simmetria di HTTP.

HTTP essendo un protocollo **richiesta-risposta** prevede che ci sia una componente passiva (server) ed una componente attiva (client) che fa delle richieste. In alcuni casi non è naturale pensare di andare a recuperare delle informazioni effettuando delle richieste, perchè l'informazione potrebbe essere prodotta **in un momento diverso** da quello in cui arriva la richiesta.

Immaginiamo il sample prodotto dall'acquisizione di un sensore, arriva la richiesta HTTP ma se il sample non è stato prodotto bisognerà effettuare una richiesta successivamente. Per rendere l'interazione tra client e server, **permettendo anche al server di promuovere la connessione**, si è pensato di introdurre un canale di comunicazione **full duplex** tra due componenti applicative molto diffuse: web browser e server web.

L'idea è di creare una connessione TCP da browser a server web e di usare quella connessione TCP per la comunicazione, che potrebbe anche essere promossa dal server.

Come gestire le web socket?

Siccome abbiamo la necessità di programmare il comportamento del browser, per fare in modo che esso attivi la connessione con il server, dobbiamo ricorrere al linguaggio di scripting JS.

Sono disponibili degli oggetti che possiamo usare per effettuare delle operazioni: l'oggetto che usiamo è di tipo **WebSocket()**

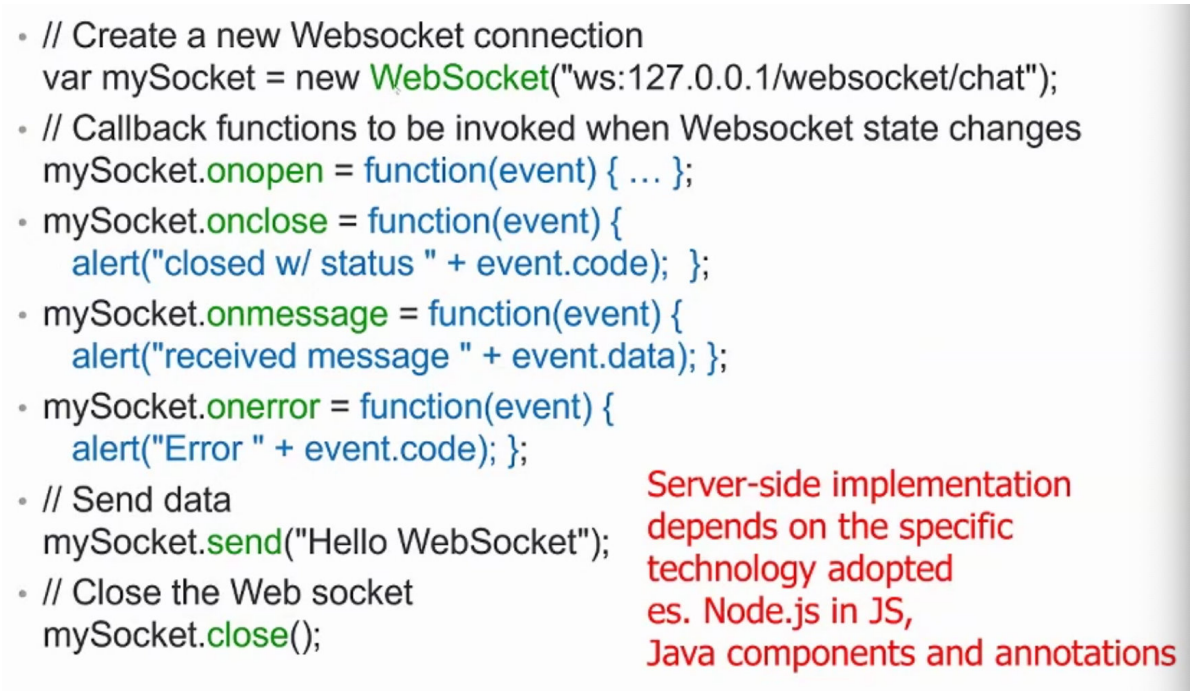
```
var ws = new WebSocket("ws://example.com/foobar");
ws.onmessage = function(event) { /* some code */ }
ws.send("Hello World");
```

Questo oggetto ci serve per realizzare una connessione con un web server, e poi usarlo per la comunicazione in due versi.

`onmessage` è l'evento di ricezione di un messaggio: quando sarà ricevuto un messaggio da parte di un endpoint lato client, sarà mandata in esecuzione una funzione. Per spedire un messaggio usando una WebSocket usiamo la funzione **send("")**.

Le WebSocket permettono anche di effettuare delle **connessioni cross-origin**, ovvero contattare un server diverso da quello che ci ha fornito il server.

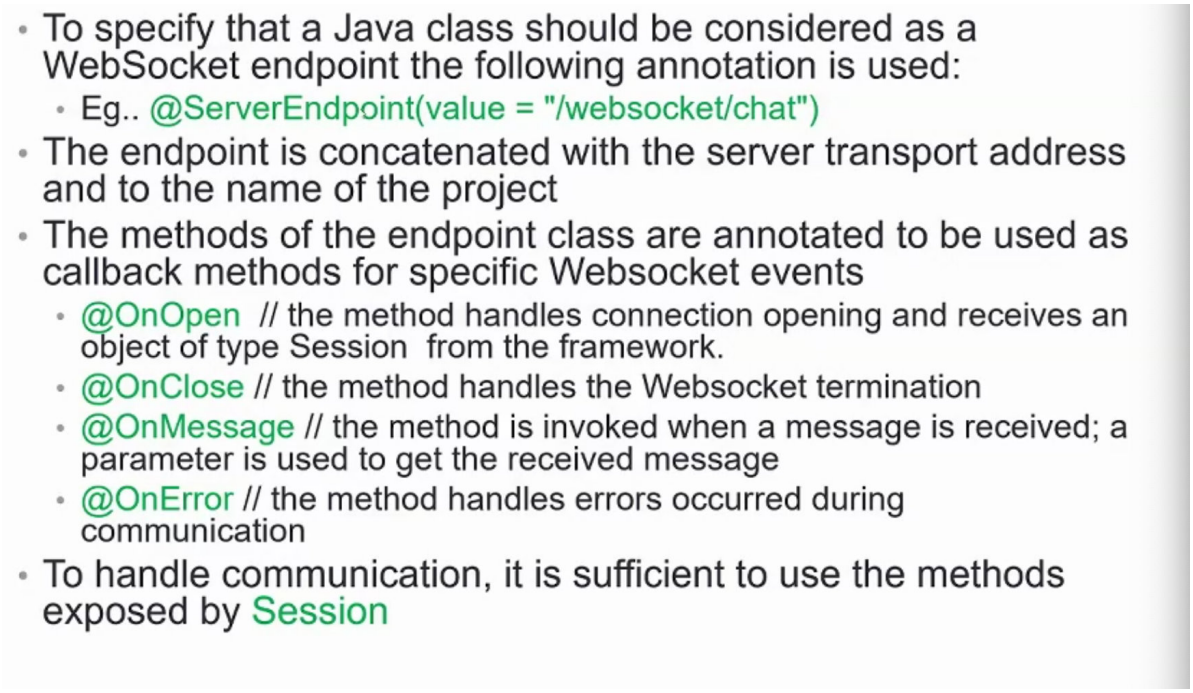
API completa



Notiamo che la differenza con la programmazione a basso livello (C) la programmazione è **orientata agli eventi**.

Lato server

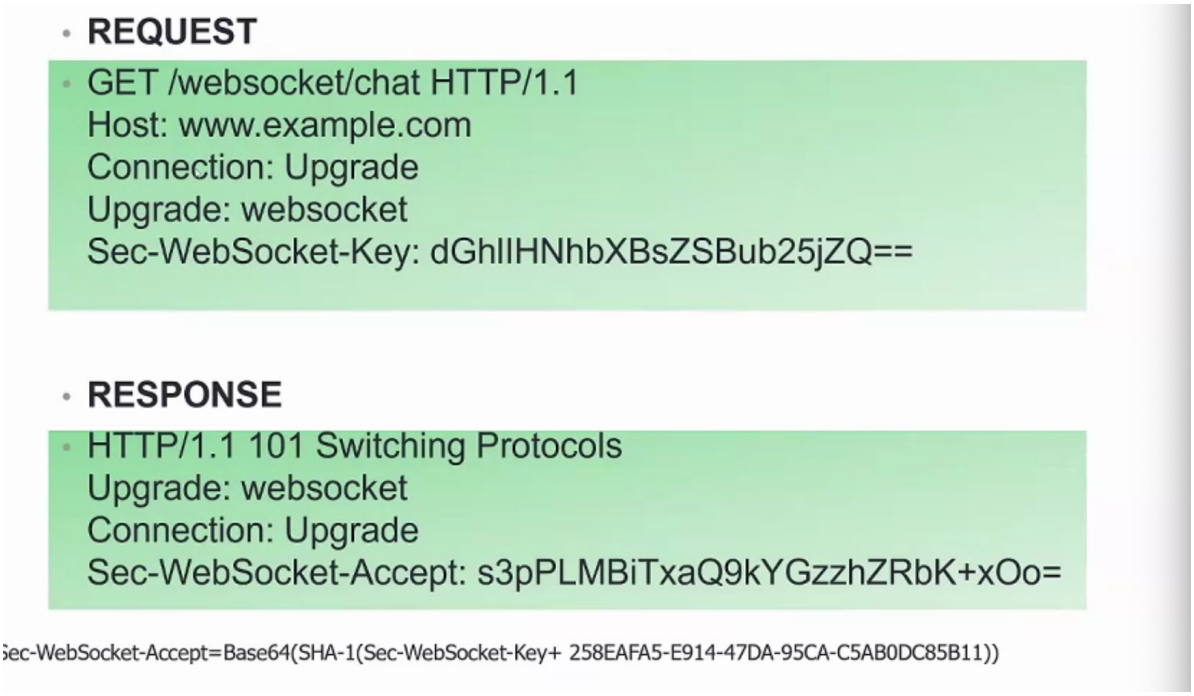
Se usiamo un server Java in ambiente Tomcat (per implementare il server), possiamo usare ancora una volta delle annotazioni:



Usiamo in particolare l'annotazione **@ServerEndpoint()** per etichettare una classe che raggruppa i metodi che diventeranno metodi di callback, ovvero metodi associati agli eventi come quelli visti lato client. Per dichiarare una classe ServerEndpoint andiamo a specificare anche qual è l'endpoint che vogliamo sia associato per operare con oggetti di quella classe.

WebSocket - HandShake

Le WebSocket sono utilizzabili in contesto web, e sono impiegate a partire da un'interazione HTTP; Quindi il browser che vuole interagire con il server con il protocollo HTTP usa tale protocollo per chiedere al server di cambiare da HTTP ad un protocollo di più basso livello, e di usare un canale di comunicazione TCP. Un messaggio proveniente dal browser all'atto di creazione di una Websocket contiene sostanzialmente ciò:



Come nel caso di HTTP/2 abbiamo il campo **connection** nel quale viene specificato che si vuole **un cambio di protocollo**. Il client indica anche una chiave per gestire la connessione che sarà usata dal server per fornire al client un input per l'avvenuta accettazione della richiesta.

Per quanto riguarda il messaggio di risposta, il messaggio di risposta è un codice di classe 1, che dice al client che si sta effettuando **uno switch di protocollo**. Il server informa quindi il client sulla possibilità o meno di continuare in modalità WebSocket (**se possiamo cambiare il protocollo o no**).

In questo caso il server accetta.

Esempio WebSocket

Creiamo ancora una volte un progetto di tipo DynamicProject (eclipse).

Nella componente client dobbiamo usare la libreria JS per la creazione di una WebSocket mentre nella componente server usiamo le annotazioni viste poc'anzi.

Per programmare il browser usiamo un documento HTML esteso con delle funzioni JS. La parte JS riportata nell'esempio è stata collocata nella sezione Head con i tag <script>. Il file HTML usato è chiamato chat.html

Body del documento

Per definire il body dobbiamo pensare alle cose da prevedere nell'interfaccia utente: dobbiamo poter immettere dei dati (testo) e dobbiamo poter visualizzare un output.
L'input usato è di tipo text e lo specifichiamo con **type="text"**.
Abbiamo inoltre due innestati che contengono il testo che verrà restituito dal client:

```
<body>
    <p>
        <input type="text" placeholder="insert text and enter to send a message" id="chat" />
    </p>
    <div id="console-container">
        <div id="console"> </div>
    </div>
</body>
```

Attribute : id
Data Type : ID

Script del documento

```
<script type="application/javascript">

    var console = {};

    console.write = function(message) {
        var p = document.createElement('p');
        p.innerHTML = message;
        document.getElementById('console').appendChild(p);
    };


    var chat = {};

    chat.socket = null;

    chat.sendMessage = function() {
        var message = document.getElementById('chat').value;
        if (message != '') {
            chat.socket.send(message);
            document.getElementById('chat').value = '';
        }
    };

    chat.connect = function(host) {

        chat.socket = new WebSocket(host);
```

Creiamo degli oggetti di supporto: **console e chat** (creazione dinamica) ed andiamo ad associare a questi oggetti delle proprietà con relativi valori; la proprietà **write** è di tipo function che riceve un parametro message. Questa funzione gestisce la console (output), quindi quando voglio visualizzare del testo, questo viene visualizzato all'interno del campo con **id=console** del codice HTML.
Creiamo quindi un elemento dinamicamente di tipo paragrafo e vi inseriamo all'interno il messaggio passato come parametro (effettuiamo un append, che non sostituisce il testo precedente).

Abbiamo un secondo oggetto: **chat**; aggiungiamo inizialmente una proprietà socket null, e successivamente aggiungiamo diverse proprietà di tipo funzione:

```
chat.sendMessage = function() {
    var message = document.getElementById('chat').value;
    if (message != '') {
        chat.socket.send(message);
        document.getElementById('chat').value = '';
    }
};
```

sendMessage viene usata per spedire dei messaggi; quando viene invocata prende il valore dal campo con id chat e lo invia; successivamente azzera il campo.

```
chat.connect = function(host) {

    chat.socket = new WebSocket(host);

    chat.socket.onopen = function () {
        console.write('Info: the connection is now active.');
        document.getElementById('chat').onkeydown = function(event) {
            if (event.keyCode == 13) {
                chat.sendMessage();
            }
        };
    };

    chat.socket.onclose = function () {
        document.getElementById('chat').onkeydown = null;
        console.write('Info: connection has been closed.');
    };

    chat.socket.onmessage = function (message) {
        console.write(message.data);
    };
};
```

L'altra funzione è **connect** che accetta come parametro l'host con cui vogliamo effettuare la comunicazione. All'interno della funzione viene creata la socket (**WebSocket**) e creiamo la connessione con l'host. Con `chat.socket.onopen = function(){...}` associamo delle proprietà di callback sulla socket, in particolare quando viene aperta-chiusa-ricezione di un messaggio.

Quando quindi avviene un evento, viene eseguito il relativo codice.

Apertura della socket

In fase di apertura c'è una scrittura sulla console (usando il metodo write visto prima) indicando che la connessione è stata aperta. Successivamente controlliamo ogni volta che viene premuto un tasto, se questo è il tasto di invio; usiamo **.onkeydown** per "leggere l'evento", e se il tasto digitato è invio, inviamo il messaggio (grazie alla funzione send).

Chiusura della socket

Quando viene chiusa la socket viene eseguito questo codice che comunica all'utente che la connessione è stata chiusa.

Ricezione di un messaggio

Quando viene ricevuto un messaggio da parte del server, usiamo la funzione console.write() (vista prima) che visualizza il messaggio sulla console.

Connessione effettiva

Alla fine dello script JS, andiamo a realizzare la connessione con l'host comunicando l'URI, andando ad invocare la funzione dell'oggetto **chat**.

Lato server

Lato server facciamo qualcosa di analogo usando le annotazioni viste prima:

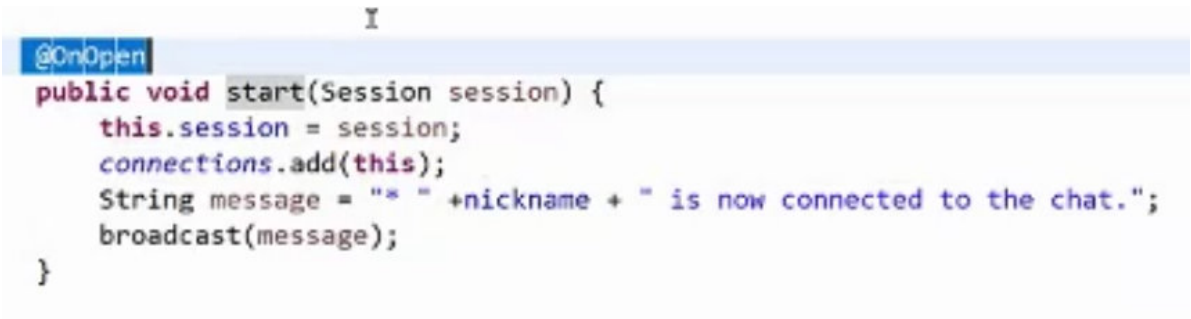
```
@ServerEndpoint(value = "/websocket/chat")
public class ChatServer {

    private static final String GUEST_PREFIX = "Guest";
    private static int connectionIds;
    private static Set<ChatServer> connections = new CopyOnWriteArraySet<>();

    private final String nickname;
    private Session session;

    public ChatServer() {
        nickname = GUEST_PREFIX + "-" + connectionIds++;
    }

    @OnOpen
    public void start(Session session) {
        this.session = session;
        connections.add(this);
        String message = "* " +nickname + " is now connected to the chat.";
        broadcast(message);
    }
```

Abbiamo la classe **ChatServer** dove troviamo una serie di metodi annotati con le annotazioni viste prima, ad esempio abbiamo il metodo receive:

```
@OnMessage
public void receive(String message) {
    String myMsg = "-> " +nickname + ": "+ message;
    broadcast(myMsg);
}
```

Con l'annotazione **@ServerEndpoint(balue = "/websocket/chat")** specifichiamo il percorso che caratterizzerà l'endpoint.

metodo start



Questo metodo che abbiamo annotato come **@OnOpen** viene eseguito ogni volta che si apre una connessione. Di conseguenza all'interno del metodo vengono eseguite alcune operazioni, tra cui il broadcast del messaggio che comunica a tutti gli endpoint che un nuovo utente si è connesso.

---

fine lezione 37