Tutorato di:

# Web Programming, Design & Usability

Tutor: Alessio Tudisco


Università di Catania

# WebSocket

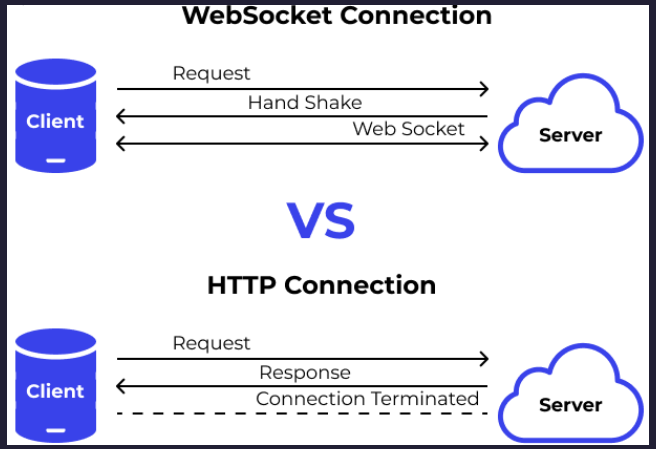
Come instaurare una comunicazione

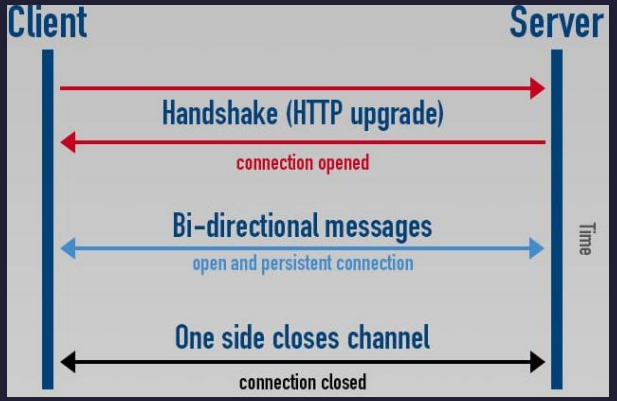
# WebSocket in breve

Le WebSocket sono un protocollo di comunicazione bidirezionale che consente l'interazione in tempo reale tra un client e un server su una connessione TCP/IP.

A differenza dei protocolli HTTP tradizionali, che seguono un modello di richiesta-risposta, le WebSocket offrono una connessione persistente tra client e server, consentendo una comunicazione più efficiente e in tempo reale.

Questo rende le WebSocket ideali per applicazioni che richiedono aggiornamenti in tempo reale, come chat online, giochi multigiocatore, flussi di dati in tempo reale e applicazioni di collaborazione.



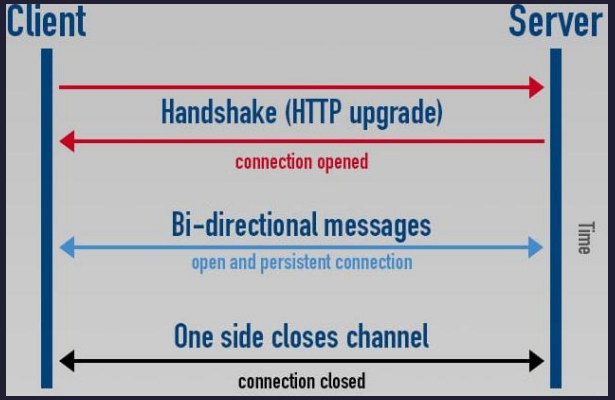



Possiamo descrivere il ciclo di vita di una WebSocket come:

1) ***Apertura della connessione***: Un client avvia una richiesta WebSocket al server utilizzando un'apposita *richiesta di upgrade del protocollo HTTP*, tramite un'intestazione speciale chiamata "**Upgrade**" con il valore "*websocket*". Il server può quindi accettare l'upgrade del protocollo e stabilire una connessione WebSocket con il client;
2) ***Handshake di connessione***: Dopo aver ricevuto la richiesta di upgrade, il server risponde con un codice di stato di conferma e stabilisce la connessione WebSocket;

Possiamo descrivere il ciclo di vita di una WebSocket come:

3) ***Scambio di messaggi***: Una volta aperta la connessione WebSocket, sia il client che il server possono inviare messaggi in qualsiasi momento senza dover attendere una richiesta dal lato opposto. I messaggi possono essere inviati come pacchetti di dati binari o come stringhe di testo;
4) ***Chiusura della connessione***: Client o server possono decidere di chiudere la connessione WebSocket. Una richiesta di chiusura viene inviata dal lato che desidera chiudere la connessione e l'altra parte risponde con una conferma di chiusura. Dopo la chiusura, la connessione WebSocket viene terminata;



# WebSocket Native JS Client

Come un client può connettersi ad una WebSocket

# Connettersi usando JS nativo

Per iniziare a comunicare, un client istanzia un oggetto WebSocket, usando come protocollo «*ws://*»:

```
const socket = new WebSocket("ws://localhost:8080");
```

*WebSocket supporta la cifratura SSL, quindi se serve una comunicazione sicura è possibile usare il protocollo «wss://»*

```
const socket = new WebSocket("wss://localhost:8080");
```

# Eventi WebSocket

Il client WebSocket nativo *offre diversi eventi* che è possibile gestire per monitorare lo stato della connessione, ricevere e inviare messaggi. Alcuni degli eventi principali sono:

- **open:** Si verifica quando la connessione WebSocket viene aperta con successo.
- **message**: Si verifica quando viene ricevuto un messaggio dal server.
- **error:** Si verifica in caso di errori durante la connessione.
- **close:** Si verifica quando la connessione WebSocket viene chiusa.

```
socket.addEventListener("open", () => {
    console.log("Connessione aperta");
});

socket.addEventListener("message", (event) => {
    console.log("Messaggio ricevuto:", event.data);
});

socket.addEventListener("error", (error) => {
    console.error("Errore di connessione:", error);
});

socket.addEventListener("close", () => {
    console.log("Connessione chiusa");
});
```

Questi eventi possono essere gestiti assegnando un *callback* tramite *addEventListener()*.
Tutti gli eventi sono gestibili anche tramite *Element Event Function* «***onEVENT = (event) => {}***»

# Invio e Ricezione dei Messaggi

Per *inviare messaggi al server*, si utilizza il metodo *send()* sull'oggetto WebSocket:

```
socket.send("Ciao, ti piacciono i Capybara?")
```

Per *ricevere messaggi dal server*, è possibile optare per:

- **Gestione dell'evento message:** definendo un callback tramite la funzione *addEventListener(), il messaggio sarà contenuto nel campo data:*

```
socket.addEventListener("message", (event) => {
    console.log("Messaggio ricevuto:", event.data);
});
```

- **Gestione del Element event function onmessage**: definendo un'espressione di funzione e assegnandola all'element event function onmessage:

```
socket.onmessage = function(event) {
    console.log("Messaggio ricevuto:", event.data);
};
```

# Chiusura della connessione

Per chiudere una connessione, il client può invocare il metodo *close()* sull'oggetto WebSocket:

```
socket.close()
```

Per comunicazioni più articolare, è possibile fornire una motivazione di chiusura tramite un codice di stato e un messaggio testuale:

```
socket.close(1001, "Il Capybara è fuggito dallo zoo")
```

# WebSocket NodeJS Server

Come creare un WebSocket lato server

# In breve...

Una volta installato il pacchetto *ws*, tramite npm, *possiamo creare la web socket affidandoci al webserver interno offerto dal pacchetto oppure usare express.*

Abbiamo a disposizione i seguenti eventi, gestibili tramite la funzione *on()*:

- **connection:** Si verifica quando un client si connette, il callback restituisce una socket rappresentante la connessione col client;
- **message**: Si verifica quando viene ricevuto un messaggio dal client;
- **close:** Si verifica quando la connessione WebSocket viene chiusa;

```
//stand alone server
const ws = require('ws')

const connection = new ws.Server({ port: 8080 })

connection.on('connection', ws => {
  ws.on('message', message => {
    console.log(`Received message => ${message}`)
  });
  ws.send('Message From Server');
});
```

```
//with express server
const express = require('express');
const ws = require('ws');

const app = express();

// Set up a headless websocket server
const wsServer = new ws.Server({ noServer: true });
wsServer.on('connection', socket => {
  socket.on('message', message => console.log(message));
});

// `server` is a vanilla Node.js HTTP server

const server = app.listen(3000);
server.on('upgrade', (request, socket, head) => {
  wsServer.handleUpgrade(request, socket, head, socket => {
    wsServer.emit('connection', socket, request);
  });
});
```

# Limitazioni delle WebSocket

- ***Compatibilità del browser***: Non tutti i browser supportano le WebSocket o potrebbero supportare solo versioni specifiche del protocollo;
- ***Firewall e proxy***: Alcuni firewall e proxy possono bloccare le connessioni WebSocket per motivi di sicurezza o configurazione. Questo potrebbe limitare l'accesso alle WebSocket da determinati ambienti di rete;
- ***Limitazioni delle connessioni simultanee***: I server WebSocket possono avere limitazioni sul numero massimo di connessioni simultanee che possono gestire. Questo potrebbe influire sulla scalabilità e sulla gestione delle connessioni in scenari ad alta concorrenza;
- ***Overhead di comunicazione***: Anche se le WebSocket sono progettate per ridurre l'overhead di comunicazione rispetto ad altre tecnologie come HTTP polling o Long Polling, ci sarà comunque un certo livello di overhead associato a ogni messaggio scambiato tra client e server;

# WebSocket con Socket.IO

Un livello di astrazione aggiuntivo per comunicare

# Socket.IO in breve

Socket.IO è una libreria JavaScript che offre un'astrazione sopra le WebSocket e altri meccanismi di trasporto per fornire un canale bidirezionale persistente tra client e server, consentendo una comunicazione in tempo reale e aggiornamenti immediati.

Alcune caratteristiche importanti sono:

- ***Compatibilità multipiattaforma***: è progettato per funzionare sia lato client (browser) che lato server (Node.js), consentendo una comunicazione bidirezionale indipendentemente dalla piattaforma utilizzata;
- ***Abilità di fallback***: offre una capacità di fallback intelligente, ovvero cerca di utilizzare la migliore opzione di trasporto disponibile (WebSocket, Server-Sent Events (SSE), ect), in base alla capacità del browser e del server.
- ***Eventi personalizzati***: è possibile definire eventi personalizzati e associare callback a essi sia lato client che lato server per consentire una comunicazione strutturata e adattabile alle tue esigenze specifiche.
- ***Stanze e namespace***: offre il concetto di **stanze** e **namespace** per organizzare i client in gruppi logici. Le stanze consentono di inviare messaggi solo a un sottoinsieme specifico di client, consentendo una comunicazione selettiva. I namespace consentono di separare la comunicazione in diverse aree tematiche o funzionalità all'interno dello stesso server WebSocket.