Node.js: comunicazione in tempo reale tra client e server

In questo tutorial vedremo come implementare la comunicazione in tempo reale tra server e client in Node.js.

Sul server invieremo tramite WebSocket al client l'output del comando ping man mano che i risultati vengono restituiti. Per fare questo dovremo utilizzare il metodo `spawn()` del modulo core `child_process` perché tale metodo dispone di eventi che possiamo gestire con funzioni di callback.

Creiamo una classe per gestire il comando ping.

```
'use strict';

const { spawn } = require('child_process');
const validator = require('validator');

class Ping {
    constructor(host, times = 1) {
        this.host = host;
        this.times = times;
    }

    validate() {
        if(!validator.isFQDN(this.host) &&
!validator.isIP(this.host)) {
            return false;
        }
        const times = this.times.toString();

        if(!validator.isInt(times)) {
```

```
            return false;
        }

        return true;
    }

    send({ ondata = function () {}, onerror =
function () {}, onclose = function () {} }) {
        if(!this.validate()) {
            throw new Error('Invalid parameters.');
        }
        const cmd = spawn('ping', ['-c', this.times,
this.host]);

        cmd.stdout.on('data', data => {
            ondata(data.toString());
        });

        cmd.stderr.on('data', data => {
            onerror(data.toString());
        });

        cmd.on('close', code => {
            onclose(code);
        });
    }
}

module.exports = Ping;
```

I parametri passati alla classe sono l'host (dominio o indirizzo IP) ed il numero di richieste da inviare. Questi parametri vengono validati e se superano la validazione vengono usati per invocare il comando ping dalla shell.

Per inviare i dati in tempo reale al client abbiamo bisogno del modulo NPM `ws` che implementa i WebSocket lato server. Questo modulo può essere utilizzato con qualsiasi instanza di un oggetto server di tipo `http` o `https`, ossia istanziato a partire da uno di questi moduli core di Node.js.

```
const express = require('express');
const ws = require('ws');
const Ping = require('./lib/Ping');
const PORT = process.env.PORT || 3000;
const app = express();

const wsServer = new ws.Server({ noServer: true });

//...

const server = app.listen(PORT);

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

In questo caso, poichè useremo l'oggetto server creato da ExpressJS, possiamo istanziare il nostro server WebSocket con il parametro `noServer` impostato su `true`.

Il client passerà al server una stringa JSON con il nome del comando da eseguire e il nome dell'host. A titolo di esempio ometteremo il parametro con cui il client potrebbe specificare il numero di ping da effettuare per concentrarci unicamente su ciò che invieremo al client tramite il metodo `send()` del nostro server.

```
//...

wsServer.on('connection', socket => {
    socket.on('message', message => {
        const messageData =
JSON.parse(message.toString());

        const { cmd, param } = messageData;
        if(cmd === 'ping') {

            const pingRequest = new Ping(param, 3);
            const callbacks = {
                ondata(data) {
                    socket.send(data);
                },
                onerror(msg) {
                    socket.send(`Error: ${msg}`);
                },
                onclose(code) {
                    socket.send(`Exit code:
${code}`);
                }
            };

            try {
                pingRequest.send(callbacks);
            } catch(err) {
                socket.send(`Error: ${err.message}`);
            }
        }
    });
```

```
});
```

Quando il server riceve dal client la stringa JSON, la trasforma in un oggetto JSON e usa la proprietà `host` per istanziare la classe `Ping` che lancerà il comando dalla shell. Ora possiamo definire e usare le tre funzioni di callback che gestiranno gli eventi ed invieranno l'output della shell al client man mano che questo viene generato usando il metodo `send()` del server WebSocket (l'oggetto `callbacks`).

Lato client dobbiamo validare l'input utente ed in caso di successo inviare i dati usando il metodo `send()` dell'oggetto WebSocket, che si istanzia nel modo seguente:

```
const socket = new
WebSocket(`ws://${location.host}`);
```

Quindi ci mettiamo in ascolto dei messaggi inviati dal server, li trasformiamo in stringhe e li usiamo per mostrare le righe di testo che man mano vengono prodotte.

```
const response = document.getElementById('ping-
response');
        const socket = new
WebSocket(`ws://${location.host}`);

        socket.addEventListener('message', event => {
            let line = document.createElement('div');
            line.innerText = event.data.toString();
            response.appendChild(line);
        });
```

Validiamo sempre i dati anche lato client e se la validazione ha successo inviamo tali dati al server.

```
const value = host.value;

            if(!validator.isFQDN(value) &&
!validator.isIP(value)) {
                host.classList.add('is-invalid');
                return false;
            }

            const data = {
              cmd: 'ping',
              param: value
            };

            socket.send(JSON.stringify(data));
```

Questa tecnica di comunicazione non si rivela utile solo nel caso delle chat, ma anche quando abbiamo la necessità di conoscere in tempo reale l'avanzamento di un'operazione sul server.