1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA APPLICATA
SCUOLA DI
SCIENZE TECNOLOGIE E FILOSOFIA DELL'INFORMAZIONE

Corso di Reti di Calcolatori
Progetto d'esame
Anno 2019-2020

Studente:
Alessandro Bernardi - Matricola: 284968

Indice

1. **Introduzione** 3
   1.1 Specifica del progetto 3
   1.2 Motivazioni 3
2. **Tecnologie utilizzate** 4
   2.1 WebSocket 4
   2.2 Ajax 4
   2.3 WebSocket vs Ajax 5
3. **Tecnologie per lo sviluppo** 6
   3.1 Node.js 6
   3.2 Express 6
   3.3 Socket.io 7
   3.4 moment-timezone 7
4. **Ambiente di sviluppo** 8
5. **Struttura del progetto** 10
6. **Implementazione del programma** 11
   6.1 Server 11
   6.2 Client web 12
7. **Messa online del progetto** 14
8. **Conclusioni** 15

1. Introduzione

1.1 Specifica del progetto

Si vuole creare un applicativo web, che permetta a più utenti connessi alla rete internet di poter scambiare messaggi.
Gli utenti saranno anonimi e sceglieranno un nome utente per identificarsi tra gli altri utenti.
Gli utenti sceglieranno inoltre una stanza, o tra quelle suggerite o una da loro inventata, per scambiare messaggi solo con le persone connesse nella medesima stanza.
Appena ci si connetterà alla stanza, si aprirà una connessione websocket con il server per lo scambio di messaggi.

1.2 Motivazioni

Si è scelto questo progetto per approfondire il paradigma della programmazione ad eventi, poco usato a livello didattico, e ad approfondire la questione *WebSocket*, un protocollo relativamente giovane, in grande espansione e supportato da molti browser, tra cui: *Google Chrome, Mozilla Firefox, Safari, Microsoft Edge*, che permette lo sviluppo di applicativi web *real-time* grazie a canali di comunicazione full-duplex, a differenza dei più vecchi *socket*.

2. Tecnologie utilizzate

2.1 WebSocket

WebSocket è un protocollo di comunicazione, che fornisce canali di comunicazione *full duplex* su una singola connessione TCP. Il protocollo WebSocket è stato standardizzato dall'IETF come RFC 6455 nel 2011.
WebSocket è distinto da HTTP, ma entrambi i protocolli si trovano al livello 7 nel modello OSI e dipendono dal TCP al livello 4.
Secondo RFC 6455, WebSocket è progettato per funzionare su porte HTTP 80 e 443, nonchè per supportare proxy e intermediari HTTP, quindi compatibile col protocollo HTTP.
L'*handshake* di WebSocket utilizza l'*header Upgrade* di HTTP per ottenere la compatibilità, inoltre il client invia `Sec-WebSocket-Key`, un header contenente byte casuali con codifica *base64* e il server risponde con un HASH della chiave `Sec-WebSocket-Accept` nell'header. Questo per evitare che un eventuale proxy invii nuovamente una precedente conversazione memorizzata in cache.
Questo protocollo consente l'interazione tra client e server con un *overhead* inferiore rispetto alle alternative *half-duplex*, facilitando la comunicazione in tempo reale.
L'utilizzo delle porte TCP 80 o 443, in caso di connessioni crittografate, è utile in tutti gli ambienti dove viene bloccato il traffico internet non web tramite firewall.

2.2 Ajax

Acronimo di *Asynchronous JavaScript and XML*, è un insieme di tecniche per lo sviluppo web di applicazioni interattive.
Con Ajax i client web possono inviare e ricevere dati da un server asincronamente, questo perché le operazioni vengono eseguite in background senza interferire con la visualizzazione e il comportamento della pagina esistente, quindi possono creare contenuti dinamici senza dover ricaricare la pagina.

2.3 WebSocket vs Ajax

Una richiesta AJAX è costruita attorno al tipico modello HTTP. Una richiesta viene effettuata da un client e una risposta viene generata da un server. Ogni richiesta crea una nuova richiesta HTTP in background.
A differenza di AJAX, i WebSocket si basano su un modello ad eventi. Il client e il server possono emettere eventi e scambiarsi dati quando necessario.
Proprio come AJAX, WebSocket deve stabilire una connessione a un server per il flusso dei dati.
A differenza di AJAX, WebSocket stabilisce questa connessione solo una volta, quindi tutti i dati vengono inviati tramite questa connessione del protocollo WebSocket aperta, ciò significa che ogni evento inviato richiede pochissime risorse sia dal server che dal client poiché non è necessario stabilire una nuova connessione.
AJAX è preferibile per tutte le applicazioni basate su richiesta e risposta.

3. Tecnologie per lo sviluppo

Per lo sviluppo del server è stato utilizzato *Node.js* con i seguenti moduli:

- *Express.js:* framework per applicazioni web.
- *Socket.io:* libreria Javascript per applicazioni web in tempo reale.
- *moment-timezone:* libreria JavaScript la formattazione delle date.

3.1 Node.js

Node.js è un *runtime* di JavaScript costruito sul motore *JavaScript V8* di Google Chrome, open source ed orientato agli eventi.
Node.js consente di utilizzare JavaScript per scrivere codice eseguibile a lato server, cosa prima non possibile in quanto JavaScript nasce come linguaggio client interpretato da un browser.
L'architettura orientata agli eventi rende possibile l'I/O asincrono. Questo design punta ad ottimizzare il *throughput* e la scalabilità nelle applicazioni web con molte operazioni *I/O intensive*, è inoltre ottimo per applicazioni web real-time.
Il modello di networking ad eventi è ritenuto più efficiente in situazioni critiche con un elevato traffico di rete, in quanto il programma può rimanere in uno stato di *sleep* fino alla notifica del sistema operativo del verificarsi di determinati eventi e solo dopo può eseguire delle funzioni di *callback*.

3.2 Express

Express è un framework per applicazioni web per Node.js e viene installato come un qualunque altro modulo.
È stato progettato per creare web application e API ed ormai definito il server framework standard per Node.js.
Express.js fondamentalmente aiuta a gestire il *back-end*, dai percorsi, alla gestione delle richieste e delle viste.

3.3 Socket.io

Socket.IO è una libreria Javascript per applicazioni web in tempo reale. Comprende una comunicazione bidirezionale realtime tra i web client e i server. È formata da due parti: una libreria lato client che gira sul browser e una libreria lato server per Node.js. Entrambi i componenti hanno la stessa struttura API. Come Node.js, è orientata agli eventi.
A differenza della libreria ufficiale *WebSocket* permette di inviare messaggi specificando un nome di un evento, aiuta la gestione dei client connessi senza mantenere una struttura apposita per tutte le connessioni.
Socket.IO non è basato su WebSocket, ma utilizza questa tecnologia solo quando è disponibile, in altri casi avvia una connessione AJAX al server, che viene successivamente aggiornato sui browser che supportano WebSocket.
Anche se, ormai, tutti i browser più popolari possiedono un supporto al protocollo WebSocket, grazie a questa libreria anche i browser non aggiornati o più datati, rendono possibile l'utilizzo della piattaforma.

3.4 moment-timezone

L'uso della libreria *moment-timezone* permette di visualizzare date e ore, in formato stringa, di un qualsiasi fuso orario.
Viene utilizzato per aggiungere l'ora di ricezione del nuovo messaggio da parte del server, per poter mostrare l'orario del messaggio nella parte client.

4. Ambiente di sviluppo

Per lo sviluppo del progetto è stato utilizzato il *sistema di controllo di versione distribuito Git*, per potere tenere traccia delle varie versioni del programma, come piattaforma di supporto a Git è stato scelto *GitHub*.

Come prima cosa è stata creata una nuova repository pubblica e poi è stata clonata in locale, nella nuova directory generata è stato inizializzato il programma Node.js con il seguente comando:

```
npm init
```

Eseguendo questo comando, il gestore di pacchetti per Node.js (npm), chiederà informazioni riguardanti il nuovo progetto: *nome, descrizione, versione, autore e file di partenza*.
Con i dati ricevuti in input e letti dalla cartella *.git*, il programma crea automaticamente il file `package.json`, un file in formato JSON contenente vari metadati rilevanti per il progetto. Questo file viene utilizzato per fornire informazioni a *npm* che gli consentono di identificare il progetto e gestire le sue dipendenze.

Per installare i vari moduli è stato eseguito il comando:

```
npm install express socket.io moment-timezone --save
```

Così facendo *npm*, aggiorna il file `package.json` aggiungendo le dipendenze per il funzionamento del progetto, crea una cartella chiamata `node-modules`, dove sono presenti i sorgenti dei moduli installati e crea il file `package-lock.json` che descrive l'albero esatto che è stato generato, in modo che le installazioni successive siano in grado di generare alberi identici, indipendentemente dagli aggiornamenti di dipendenza intermedi.

Per una comodità di sviluppo è stato installato anche *nodemon*, uno strumento che aiuta a sviluppare applicazioni basate su node.js riavviando automaticamente l'applicazione quando vengono rilevate modifiche ai file nella directory e viene installata con il seguente comando:

```
npm install nodemon --save-dev
```

Così facendo si precisa che non è strettamente necessario per l'esecuzione, ma può servire per lo sviluppo e il testing.

A questo punto è stata aggiunto al file `package.json` il seguente attributo:

```
"scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
},
```

In questo modo sarà più semplice avviare il programma normalmente o in modalità sviluppo con uno dei seguenti comandi:

```
npm run start
npm run dev
```

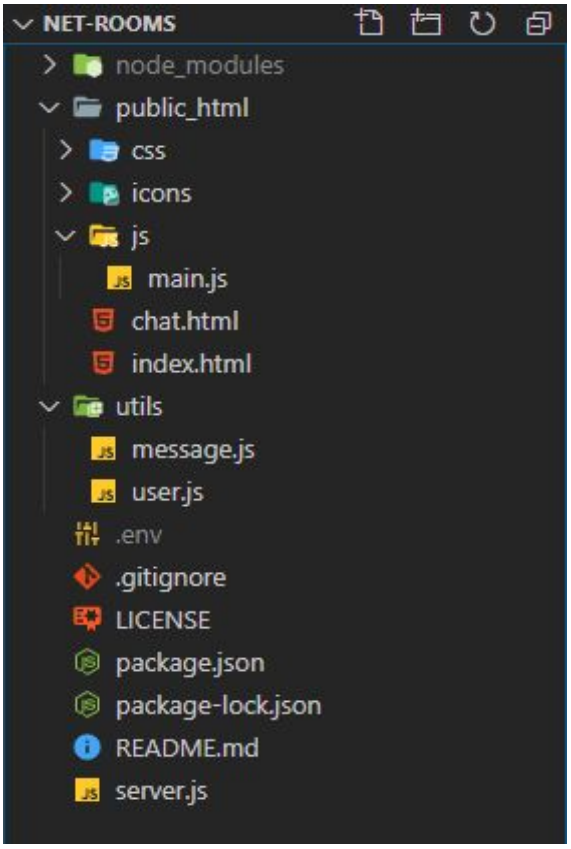
Come ultima cosa per la preparazione dell'ambiente è stato aggiunto al file `.gitignore` la cartella, che contiene i moduli, `node_modules`, non necessaria all'upload sul repository GitHub, in quanto facilmente generabile dopo aver clonato la repository tramite il comando:

```
npm install
```

Così facendo viene letto il contenuto dei file `package.json` e `package-lock.json` e viene generato lo stesso identico albero delle dipendenze utilizzato in locale al momento dello sviluppo.

5. Struttura del progetto

Nell'immagine seguente è mostrata la struttura delle sotto directory della cartella principale del progetto, `net-rooms`.



Come mostrato nell'immagine la struttura del progetto è molto semplice, nella cartella principale oltre ai file di configurazione per *npm* e la cartella `node_modules`, è presente il file di partenza `server.js`, colui che verrà avviato per l'esecuzione del server web.

Per rendere più semplice il modello del programma, le funzioni utilizzate da `server.js` per la gestione degli utenti connessi e dei messaggi, sono stati creati due moduli presenti nella cartella `utils`, chiamati rispettivamente `user.js` e `message.js`.

La cartella chiamata `public_html`, è la cartella statica dove vengono inseriti tutti i file del sito web, all'interno si trova anche la cartella `js` che contiene gli script per la gestione delle connessioni WebSocket lato client.

6. Implementazione del programma

6.1 Server

Il server è scritto in Node.js e sfrutta i moduli precedentemente descritti.
Per prima cosa cosa sono stati importati i moduli e funzioni sia quelli installati esternamente tramite *npm*, sia quelli scritti ad hoc per l'implementazione del progetto.

In seguito è stato creato un oggetto *server*, utilizzando Express, ed avviato in ascolto su una determinata porta, la scelta della porta avviene tramite un meccanismo che determina la presenza o meno di una variabile d'ambiente con l'apposito valore da attribuire alla porta, in caso di mancanza il server viene messo in ascolto sulla porta 8080, usata spesso per far girare un server web come utente non amministratore.

```
const PORT      = process.env.PORT || 8080;
const listener  = server.listen(PORT, () => {
    console.log(`The server is listening on port:
                ${listener.address().port}`)
});
```

Per definire la cartella statica contenente le pagine web, o in generale il *front-end* è stato usato un Express come *middleware*, ovvero come un software che funge da intermediario tra strutture e programmi, permettendone la comunicazione.
Express ricerca i file relativi alla cartella statica, pertanto il nome della cartella statica non è parte dell'URL.

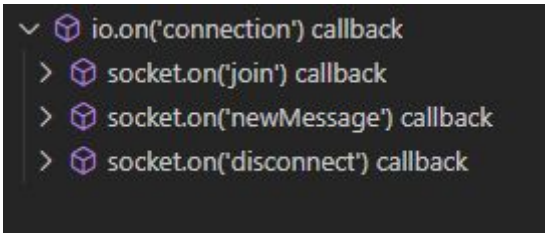
Per la gestione delle connessioni è stato creato un oggetto di Socket.io, chiamato *io* e viene messo in ascolto per una nuova connessione.
Su un evento di connessione *(connection)*, viene eseguita una funzione di callback, che ha come parametro un oggetto *socket*, l'uso principale di questo oggetto è quello di metterlo in ascolto per altri eventi.
Ogni connessione ha un socket in ascolto differente, in modo tale da poter distinguere le comunicazioni, i possibili eventi sono:

- `join`: generato da un nuovo client che si connette ad una stanza, il client invierà un oggetto con il suo nome utente e la stanza scelta. La funzione di callback aggiunge l'utente all'array degli utenti connessi, sottoscrive il socket

al canale della stanza scelta, invia al client che si è connesso un messaggio di benvenuto, notifica tutti gli altri socket sottoscritti al canale nel nuovo utente connesso e invia gli aggiornamenti sulle informazione della stanza.

- `newMessage`: generato dall'invio di un nuovo messaggio da parte di un utente, il client invierà il messaggio in formato stringa, la funzione di callback troverà l'utente in base all'id del socket, e manderà il messaggio a tutti gli utenti connessi alla medesima stanza.
- `disconnect`: generato dalla disconnessione volontaria o meno di un client, la funzione di callback cerca l'utente tramite id del socket e lo rimuove dall'array degli utenti connessi, notifica gli utenti connessi alla stanza della disconnessione e invia gli aggiornamenti sulle informazione della stanza.

Struttura della gestione delle connessioni:



6.2 Client web

La parte web comprende sostanzialmente due pagine web, l'indice del sito usato per effettuare l'accesso ad una determinata stanza e la chat vera e propria.
Sono stati sviluppati prevalentemente mediante l'uso del framework *Bootstrap 4*, che raccoglie strumenti per la creazione di siti e applicazioni web, quali *CSS e JavaScript*.

La pagina di accesso comprende una form che presenta un campo di testo semplice per l'inserimento del nome utente e un campo di testo concatenato con un elemento `<datalist>` per la scelta della stanza. L' elemento HTML `<datalist>` contiene un insieme di elementi `<option>` che rappresentano le opzioni consigliate per la scelta della stanza.
La form reindirizza alla pagina della chat, passando i dati tramite parametri query.

La pagina della chat è la più complessa e comprende due sezioni distinte, a sinistra è presente una *sidebar* dove vengono visualizzate le informazioni della stanza (nome, lista di utenti connessi), a destra è presente la chat con in alto i messaggi e in basso una form per l'invio di nuovi messaggi.

La pagina esegue il file `main.js` presente nella cartella `js`, questo file contiene le istruzioni e le funzioni per rendere la pagina dinamica.

Per prima cosa viene creata una costante `socket`, presa dall'API di *Socket.io* tramite il path `/socket.io/socket.io.js` richiesto dal file HTML. Con questa costante è possibile utilizzare la libreria per rimanere in ascolto o inviare eventi.

Dopo aver preso i parametri query da URL, viene inviato un evento `join` con un oggetto composto da nome utente e stanza, per notificare al server l'unione alla stanza.

In seguito viene messo in ascolto `socket` sui seguenti eventi:

- `informations`: generato dal server per inviare le informazioni inerenti ad una stanza: nome, lista di utenti connessi. La funzione di callback esegue la funzione `printInfo({room, users})` spiegata in seguito.
- `message`: generato dal server per notificare l'arrivo di un nuovo messaggio nella, da parte di un utente o del server stesso *(es. messaggio di benvenuto)*. La funzione di callback esegue la funzione `printMessage(message)` spiegata in seguito.

Viene messo in ascolto l'oggetto *Element* che rappresenta la form per l'invio del messaggio in ascolto su `submit`, l'evento generato al premere del pulsante di invio del messaggio.
La funzione di callback prima di tutto esegue la funzione `preventDefault()` dell'interfaccia *Event*, questa funzione annulla l'evento se è annullabile, il che significa che l'azione predefinita che appartiene all'evento non si verificherà, in questo caso evita che la il reindirizzamento di pagina dopo aver eseguito un *submit*. In seguito viene resettato e rimesso il focus sul campo di testo, si invia poi un evento `newMessage` contenente il messaggio da inviare.

Le funzioni elencate in precedenza sono:

- `printMessage(message)`: come parametro richiede un oggetto `message` contenente nome utente, messaggio e ora. Viene creato del codice HTML per il singolo messaggio ed appeso all'interno dell'elemento che contiene l'elenco dei messaggi, per poi farlo scorrere verso il fondo per mostrare il nuovo messaggio.
- `printInfo({room, users})`: come parametro viene passato un oggetto contenente nome della stanza e lista di utenti connessi. Viene sostituito il codice HTML nella *sidebar* con le informazioni più aggiornate, per avere un elenco aggiornato in tempo reale degli utenti connessi nella stanza.

La disconnessione invia automaticamente un evento `disconnect` al server.

7. Messa online del progetto

Per mettere online la piattaforma è stato utilizzato il servizio di continuous delivery messo a disposizione gratuitamente da Heroku. Viene collegata alla propria app di Heroku un repository pubblico, l'app online viene automaticamente aggiornata ogni qualvolta venga effettuata una modifica sul branch master del repository collegato.

Per motivi di sicurezza l'app è stata tenuta in modalità *mantenimento* fino al raggiungimento di una versione più o meno stabile su cui fare testing.

La piattaforma di Heroku automaticamente installa le dipendenze descritte nei file `package.json` e `package-lock.json`, così facendo non c'è bisogno di caricare nella repository i moduli, che potrebbero appesantire l'upload e il download. Inoltre sempre nel file `package.json` sono descritti gli script di esecuzione ed Heroku può così eseguire il server senza problemi.

Come descritto in precedenza per la scelta della porta è stato implementato un meccanismo che rileva la presenza o meno di una variabile d'ambiente per la porta, Heroku dispone già di questa variabile "*PORT*" e non può essere scelta arbitrariamente.

Il progetto è stato chiamato `net-rooms` ed è online sull'indirizzo:
https://net-rooms.herokuapp.com/

8. Conclusioni

Gli obiettivi predisposti sono stati raggiunti, l'uso di Node.js ha favorito lo studio sul paradigma di programmazione ad eventi e la libreria Socket.io ha facilitato l'utilizzo e la comprensione del protocollo WebSocket.

Grazie a Node.js e il framework Express è stato semplice ed intuitivo lo sviluppo del server web, la gestione degli eventi rende molto facile e veloce l'implementazione.

Socket.io è un'ottima libreria per interfacciarsi alle comunicazioni *real-time* semplice ed intuitiva, con un carico di studio non troppo elevato è stato possibile sviluppare un software in grado di gestire più connessioni. Ottima l'API per lo sviluppo del client web, simile in tutti gli aspetti alla libreria per lo sviluppo del server.
Socket.io è un'ottima alternativa alla libreria standard *WebSocket*, permette di ottenere gli stessi risultati con un compromesso sulle prestazioni, infatti la libreria WebSocket esegue due richieste, una per la pagina HTML e una per la connessione WebSocket, invece Socket.io esegue più richieste, una per per la pagina HTML, una per richiedere all'API gli script, diverse richieste AJAX per accertarsi della possibilità di una connessione WebSocket.
Pur essendoci questa differenza nel traffico di rete può essere preferibile in quanto, nei casi in cui il browser non abbia il supporto a WebSocket, automativamente Socket.io esegue un downgrade al polling AJAX, rendendo disponibile l'applicazione web anche browser più datati.

La piattaforma Heroku pur essendo gratuita offre un buon servizio, soprattutto per le connessioni sicure, in quanto implementa il protocollo HTTPS.