Relazione Hotelier

Davide Fantasia - Mat **634105** - Reti e Laboratorio3 2023/2024

Istruzioni all’Uso
- Come Compilare
  - da file sorgente
- Come Eseguire
  - Da File Sorgente
  - Da Jar
- Comandi Possibili

Thread Attivi
- Client
- Server

Strutture Dati principali
- Client
  - primitive di sincronizzazione
- Server
  - primitive di sincronizzazione

Scelte Implementative
- Punteggio degli hotel

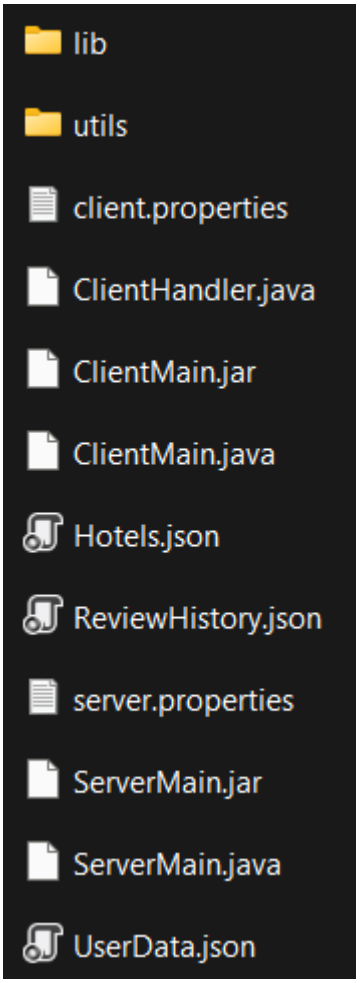
link utili

Istruzioni all’Uso

Come Compilare

da file sorgente

Per compilare i file da file sorgente è necessario avere tutti i file come da immagine.



Nella cartella *lib* è presente il jar gson.jar (versione 2.10.1); mentre nella cartella *utils* sono presenti tutti gli script e le classi di utility usate nel progetto.

I due file *.jar* non sono necessari al corretto funzionamento del progetto.

Il codice è stato scritto in JAVA8, per cui bisogna prima verificare di avere la corretta versione del JDK (download).

Il comando per compilare il server è

```
javac -cp .;./lib/* ServerMain.java
```

Se si vuole forzare la compilazione in JAVA8 pur avendo una JVM di una versione aggiornata si può mandare il comando

```
javac -cp .;./lib/* -source 1.8 -target 1.8 ServerMain.java
```

dove il flag `-cp` è il flag per impostare il “class path” del progetto, indicando al compilatore di includere la libreria gson.jar.

Analogamente per compilare il client useremo:

```
javac ClientMain.java
```

e per forzarne la compilazione in JAVA8 se si ha un versione più aggiornata della JVM si può usare

```
javac -source 1.8 -target 1.8 ClientMain.java
```

Questi comandi genereranno un file ‘.class’ per ogni classe del progetto, una volta fatto ciò il progetto è pronto ad essere eseguito.

Come Eseguire

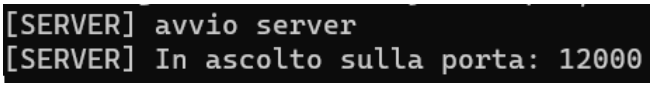
Per eseguire il progetto, sia che si tratti di source file che di JAR, bisogna tenere a mente che il Client apre una connessione con il Server, per cui se il Server è non avviato all’avvio del Client, il processo Client terminerà in modo brusco, dando un errore di questo tipo

Da File Sorgente

Dopo aver compilato i file sorgente, per eseguire il **Server** basta usare il comando

```
java -cp .;./lib/* ServerMain
```

Questo manderà in esecuzione il Server, che se avviato correttamente ci informerà dell’avvenuta accensione e della porta su cui è in ascolto.



Se si vuole cambiare la porta su cui il server sta in ascolto bisogna cambiare l’omologo campo nel server di configurazione *server.properties*, facendo lo stesso anche per il file di configurazione del client.

Il Client si esegue con l’analogo comando:

```
java ClientMain
```

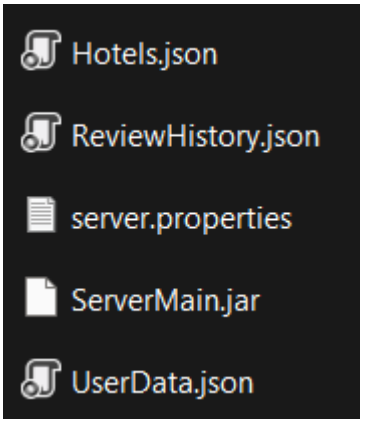
che se eseguito correttamente si dovrebbe aprire con il *Main Menu* del client:

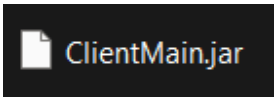
Da Jar

Requisiti

Per compilare il **Server** i file necessari sono il *ServerMain.jar*, *server.properties* e gli altri *.json*, come mostrato in figura.

Tranne il file *Hotels.json*, i file json possono essere anche vuoti.





Per compilare il **Client** invece non sono necessari altri pre-requisiti in termini di file, basta avere solo il file *.jar* e il *client.properties*.

Comandi

I comandi da usare per eseguire i file sono

- per il **Server**

```
java -jar ServerMain.jar
```

- per il **Client**

```
java -jar ClientMain.jar
```

Per entrambi, una volta eseguiti (prima il Server e poi il Client) valgono gli stessi esiti della compilazione da File Sorgente.

Comandi Possibili

Come illustra il Main Menu del Client, un utente che avvia il programma client può comunicare con il server scrivendo sul terminale i comandi che desidera con gli argomenti richiesti.

Se si sbaglia la sintassi dei comandi o si mandano comandi non registrati fra quelli disponibili viene restituito il codice di errore `[WRONG_INPUT_ERROR]`, in caso opposto, se il comando va a buon fine viene restituito il codice `[ SUCCESS ]`. L'input non è case sensitive.

IMPORTANTE: per far capire al parser del server che si sta indicando lo stesso argomento, meglio passarlo fra virgolette. Se il nome del mio hotel per esempio ha degli spazi, bisogna metterlo fra apici.

> es: se devo passare un nome di un hotel che ha degli spazi, lo metterò fra apici:
>
> Hotel Cagliari 1 $\rightarrow$ "Hotel Cagliari 1"

I comandi disponibili sono:

- `exit` effettua il logout se precedentemente loggato e chiude l'esecuzione del programma in maniera controllata.
- `register < username > < password >` dove `< username >` e `< password >` sono lo username e la password con le quali l'utente si vuole registrare. La password (che può essere alfanumerica in formato UTF-8). La password è registrata sul file *UserData.json* dopo essere stata crittografata tramite una funzione Hash One-Way crittograficamente sicura (SHA-256) il cui metodo è implementato nel file *Hashing.java*, questo permette di non mantenere le password salvate in chiaro nel file degli user.
- `login < username > < password >` dove `< username >` e `< password >` sono lo username e la password con le quali l'utente si è registrato. Se la password risulta sbagliata o l'utente non risulta presente viene restituito l'errore `[WRONG_PASSWORD_ERROR]`.

  Il Log-In inoltre 'iscrive' l'utente al canale multicast per ricevere le notifiche sul Ranking dei vari hotel. Il Log-Out e il comando di Exit automaticamente fanno

uscire l’utente dal canale.

A scopo di test è già presente un profilo “admin” la cui password è “admin”

- `logout` effettua il log-out dal profilo se si ha eseguito il log-in, altrimenti restituisce il codice di errore `[NOT_LOGGEDIN_ERROR]`.
- `searchAllHotel < city >` Ricerca tutti gli hotel fra l’elenco di hotel disponibili in una determinata città, dove `< city >` è il nome della città. Se la città non è fra quelle disponibili viene restituito l’errore `[NO_SUCH_CITY_ERROR]`. Se il comando va a buon fine viene restituita tutta la lista di Hotel in città con relative informazioni.
- `searchHotel "< nome hotel >" < city >` Cerca uno specifico Hotel in una data città `< city >` con un dato nome `< nome hotel >` restituendo le informazioni su quel dato Hotel. Se la città non è presente viene dato lo stesso codice di errore di prima, mentre se l’hotel non risulta nella lista degli hotel in quella città viene restituito `[NO_SUCH_HOTEL_ERROR]`.
- `insertReview "< nome hotel >" < city > < Global Rate > < cleaning > < position > < services >< quality >` Permette di inserire una nuova recensione recensione, i punteggi inseriti devono andare da 0 a 5, se vengono messi come maggiori di 5 vengono impostati a 5 e analogamente se sono minori di 0, vengono impostati a 0. Se l’hotel o la città non vengono trovato vengono restituiti gli stessi tipi di errori del comando *SearchHotel*. Questo comando **è usabile solo se si ha fatto il log-in**, altrimenti viene restituito l’errore `[NOT_LOGGEDIN_ERROR]`
- `showMyBadge` permette al client di sapere qual è l’ultimo badge ottenuto per il suo contributo al servizio. Ogni 5 recensioni viene dato un nuovo badge, ci sono 5 badge in totale:
  - (1) Recensore
  - (2) Recensore Esperto
  - (3) Contributore
  - (4) Contributore Esperto
  - (5) Contributore Super

  Quello fra parentesi è anche il valore di ciascuno badge, che viene poi usato come peso per la recensione dell’utente.

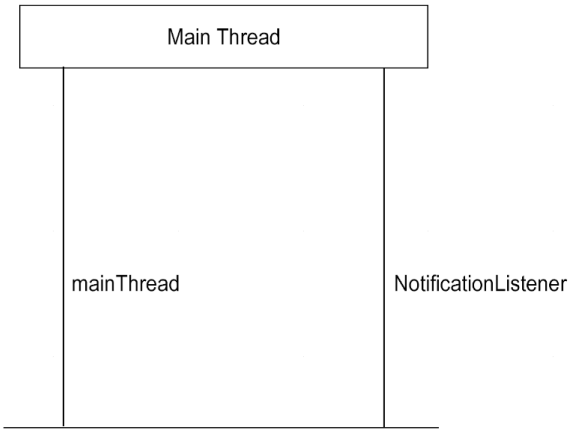
```
//calcolo delle nuove votazioni pesate sulla nuova recensione
userRating.forEach((k,v)->{
    //il valore delle vecchie recensioni è pesato con il numero di recensioni che ha
    //messo in media con la recensione dell'utente pesata con il suo badge
    double newValue = (numberOfRev*ratings.get(k))+(badge_value*v);
    newValue = newValue/(numberOfRev+badge_value);
    newRating.put(k, newValue);
});
searchedHotel.setRatings(userRating);
```

Thread Attivi

Client

La struttura del Client è pensata per essere molto più leggera e veloce del Server, per cui l'unica azione che fa il Client è mandare stringhe e ricevere risposte, e in un thread a parte riceve le notifiche del Ranking.

- MainThread: inizializza il NotificationListener Thread e tutte le variabili necessarie. In loop prende in Input da tastiera i comandi dell'utente e li inoltra al Server; si mette poi in ascolto sulla porta letta dal config file (*client.properties*) tramite connessione TCP col server. In base al comando e all'esito del comando vengono eseguiti gli effetti collaterali del comando (es: il comando 'exit' lato server forza il logout, ma lato client fa anche terminare l'esecuzione dell'applicativo).

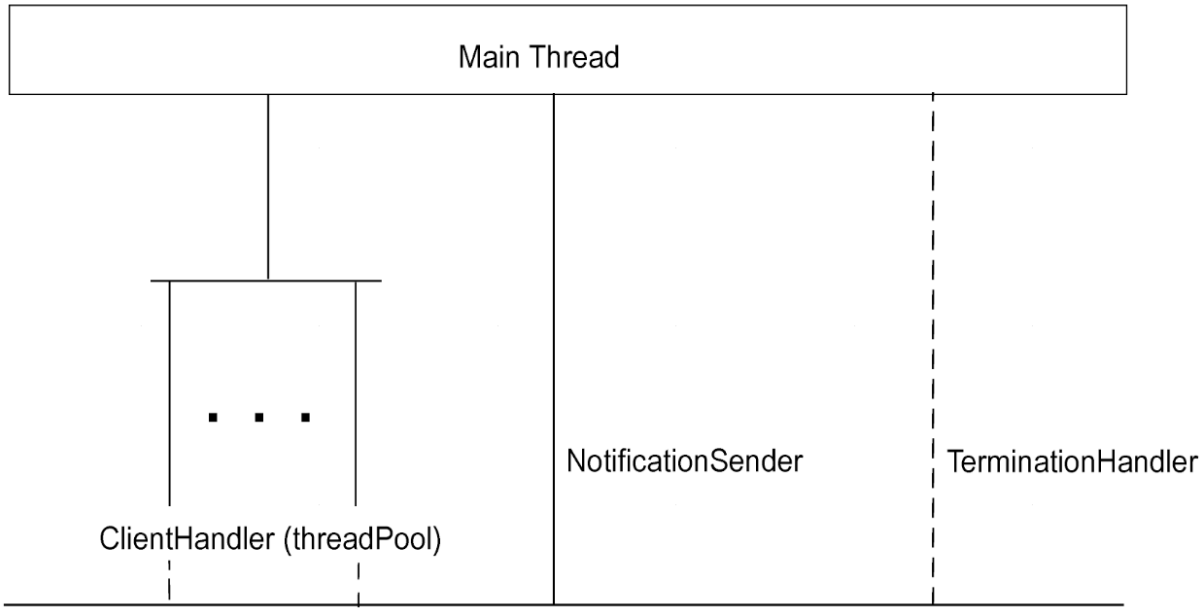


- NotificationListener: si occupa di rimanere in ascolto sulla porta del multicast, in attesa di nuove notifiche da mostrare a schermo.

Server

Il server ha principalmente 3 tipi di thread:

- ClientHandler: è un pool di thread, ogni thread viene istanziato quando si apre una nuova connessione con un nuovo utente con la classe `ClientHandler` presente nel file *lib/ClientHandler.java*, il metodo *run* del thread va a gestire in loop i comandi in ingressi dal socket del client.
- NotificationSender: è il thread dedicato al calcolo del ranking locale, ogni 6 secondi ($\frac{maxDelay}{10}$) esegue il calcolo e verifica se rispetto alla top degli hotel che era stata salvata in precedenza, ci sono aggiornamenti, ed in caso usa i metodi dati dalla classe `NotificationManager.java` per comunicare tramite pacchetti *UDP multicast* ai client gli aggiornamenti nella top.
- TerminationHandler: Non è un vero e proprio thread attivo, in quanto funge da *Shutdown Hook*, ha lo scopo di entrare in funzione alla chiusura del server, principalmente quando si chiude con `^C`, una volta attivo va a chiudere in maniera controllata tutte le risorse lasciate aperte a RunTime, andando chiudere anche il Thread di NotificationSender.



Strutture Dati principali

Client

Il client per sua natura non usa struttura dati elaborate

primitive di sincronizzazione

Si è scelto di implementare la stampa su console in maniera sincrona attraverso i metodi della classe `ConsolePrinter`, nel file *ConsolePrinter.java*, in quanto può accadere che più entità (magari la risposta dal server arriva in contemporanea alla notifica di aggiornamento dei TopHotel) provino a stampare a schermo contemporaneamente, usando il metodo `synchronized void printToConsole(String msg)` evitiamo la *race condition*.

Server

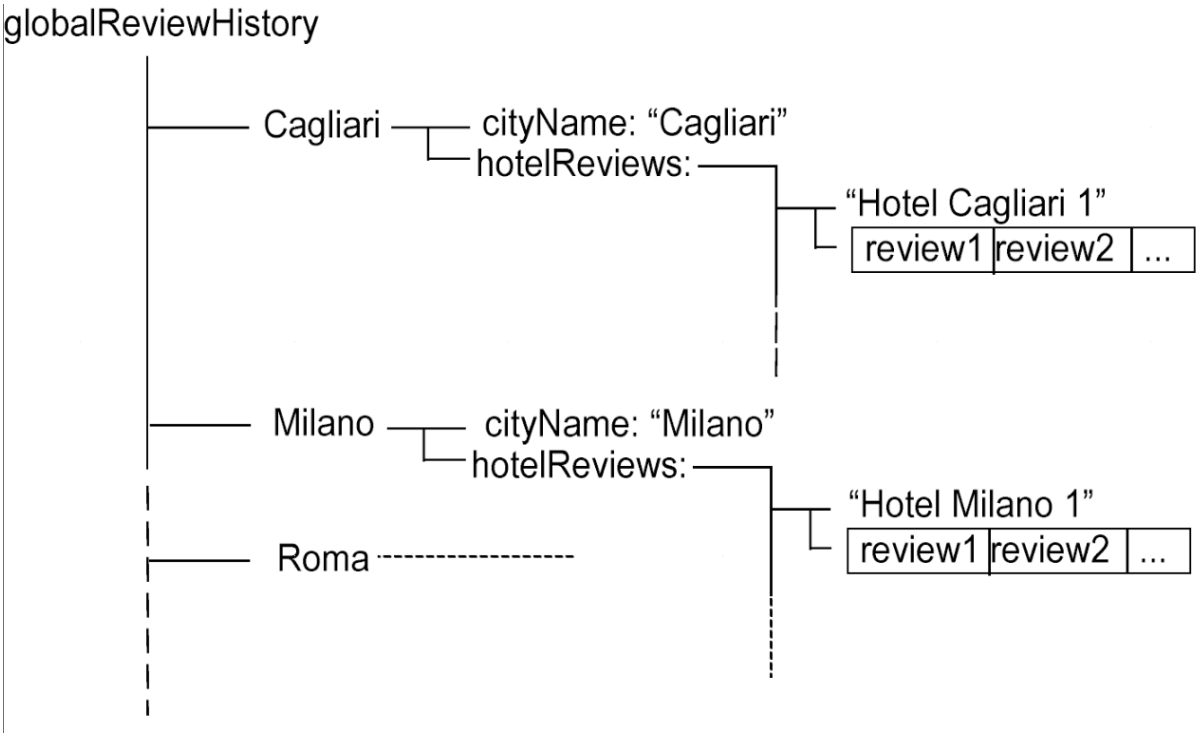
Il server a differenza del client ha diverse strutture dati elaborate da discutere

- La classe User descritta nel file User.java mantiene un oggetto statico `listOfUsers`, che descrive una lista di *User* (*Hashmap<String, User>*) che mappa lo username all'istanza di User. La classe offre alcuni metodi statici, tra cui quello per il login e la registrazione, inoltre il metodo statico (chiamato solo internamente alla classe e in fase di chiusura) di salvataggio dati su file è di tipo synchronized per evitare che più entità contemporaneamente possano provare a scrivere su file.
- La classe Ranking nel file Ranking.java descrive invece la classifica dei migliori Hotel. Il thread lanciato nel file ServerMain.java si basa su questi metodi che gli restituiscono i relativi migliori hotel. La classe mantiene a runtime le informazioni sulla città ed il suo miglior hotel in una `HashMap<String, Hotel>`.
- La classe ReviewHistoryManager nel file ReviewHistoryManager.java si occupa di mantenere, salvare su file, leggere da file e restituire informazioni sullo storico delle recensioni di ogni hotel. È una classe singleton;

```
private static class Review {
        private String username;
        private Double globalScore;
        private Map<String, Double> userRating;
  ...
}
```

l'oggetto `globalReviewHistory` è un'istanza di `HashMap<String, CityReviewHistory>` e rappresenta le recensioni di tutte gli hotel di tutte le città. La HashMap ha come chiavi una stringa che rappresenta la città, e come valore un oggetto di tipo "CityReviewHistory", che rappresenta lo storico delle recensioni di tutti gli hotel situati

in quella città, infatti contiene una hashmap, la cui chiave (una stringa) sono il nome degli hotel e il valore un CopyOnWriteArray<Review> che rappresenta lo storico di recensioni in ordine di inserimento.



primitive di sincronizzazione

Tutte le strutture usate per il mantenimento dei dati sono MT-safe (CopyOnWriteArrayList) e i metodi che si usano per la scrittura su file sono metodi *synchronized*.

Scelte Implementative

Punteggio degli hotel

Per il calcolo del punteggio degli Hotel si ha optato di scegliere gli Hotel con Rate più alto. I cui singoli punteggi vengono calcolati come:

$$newScore_i = \frac{(Score_{user_i} \times BadgeValue_{user}) + (Score_{prev_i} \times nRec)}{BadgeValue_{user} + nRec}$$

Ovvero una media pesata, dove il nuovo Voto dell’utente ($Score_{user}$) viene pesato con il valore del suo badge ($BadgeValue_{user} \in [1, 5] \subseteq \mathbb{Z}$) mentre il voto attuale viene pesato il numero di recensioni ($nRec$).

link utili

- JDK - https://www.oracle.com/it/java/technologies/javase/javase8-archive-downloads.html
- gson.jar - https://repo1.maven.org/maven2/com/google/code/gson/gson/2.10.1/
- file sorgenti e JAR - https://github.com/DavideFantasia/HOTELIER