UNIVERSITÀ DI PISA

Progetto Laboratorio Di Reti 2022/2023

WORDLE

Daniele Zeolla mat. 548824

Prof.ssa Laura Emilia Maria Ricci

INDICE

1. Descrizione del gioco

Wordle e' un famoso gioco online in cui i giocatori devono indovinare la parola del giorno in un numero limitato di tentativi.
La parola segreta di 10 lettere da indovinare viene estratta in modo casuale da un dizionario del server.
Dopo ogni tentativo il gioco fornisce al giocatore un feedback sulle lettere che compongono la parola da indovinare: viene suggerito (solitamente sotto forma di colori delle lettere) le lettere nella posizione corretta, le lettere nella posizione sbagliata e le lettere completamente sbagliate.
Obiettivo del gioco è quello di indovinare la parola segreta con il minor numero di tentativi possibili.
La parola da indovinare viene aggiornata periodicamente dal server.

2. Struttura del progetto

Per garantire una migliore leggibilità e separazione del codice, il progetto è stato suddiviso in 3 cartelle principali:

- **src/java/server/**
- **src/java/client/**
- **src/java/common/**

Server/ e **client/** contengono rispettivamente il codice del server e del client.
La directory **common/** contiene tutte quelle classi e utility comuni che vengono utilizzati dai due applicativi.
Questo permette di includere nei pacchetti **jar** solamente le classi strettamente necessarie e quindi di ridurre al minimo indispensabile la loro dimensione.

3. Build, esecuzione e dipendenze

Per la gestione delle dipendenze e la creazione dei pacchetti **.jar** è stato scelto di utilizzare **maven**, un noto strumento che a partire dal file di configurazione (**pom.xml**) gestisce in modo automatico ed indipendente le dipendenze del progetto e la generazione dei pacchetti jar di client e server.

3.1 Build

Dopo aver installato maven sul proprio sistema (vedere qui) è possibile installare le dipendenze, compilare il codice e generare i file .jar con un semplice comando:

```
mvn clean install
```

Il risultato della compilazione verra' salvato nella cartella **target/.** Nello specifico il risultato della compilazione genera i seguenti file **jar:**

- **Client-jar-with-dependencies.jar**
- **Server-jar-with-dependencise.jar**

3.2 Esecuzione

I comandi necessari per la corretta esecuzione delle applicazioni client e server sono i seguenti:

```
WORDLE_CONFIG=client.config java -jar target/Client-jar-with-dependencies.jar

WORDLE_CONFIG=server.config java -jar target/Server-jar-with-dependencise.jar
```

Variabili d'ambiente:

- **WORDLE_CONFIG** -> path al file di configurazione
- **WORDLE_DEBUG** -> true/false, avvia in modalita debug (log debug)

In alternativa, per lanciare sia il server che il client utilizzando i file di configurazione predefiniti (presenti nella root del repository) è possibile utilizzare lo script ***run.sh*** con i seguenti comandi:

```
chmod +x run.sh
./run.sh server # lancia il server in modalita' debug
./run.sh client # lancia il client
```

3.3 Dipendenze

Come specificato nella sez.3.1 è stato scelto di utilizzare **maven** per la gestione dell'unica dipendenza del progetto ovvero **com.google.code.gson** (versione 2.8.9).
Questa libreria viene utilizzata all'interno del progetto per la serializzazione e la deserializzazione di oggetti JSON.

*Nota: Per completezza il file **.jar** della dipendenza e' stato allegato al progetto nella cartella src/main/resources.*

4. Implementazione

In questa sezione sono illustrati i dettagli implementativi che riguardano le applicazioni server e client.

4.1 Server

La classe principale del server e' **ServerMain** nella cartella src/main/java/server. L'interazione tra client e server viene effettuata mediante **RMI** e richieste **TCP/UDP.**

4.1.1 TCP

Il server **TCP** è stato implementato utilizzando la libreria **NIO** con gestione del multiplexing dei canali non bloccante.
La gestione delle singole richieste è stata affidata a una **threadpool**, la quale le elabora in modo parallelo e non bloccante rispetto all'arrivo di nuove richieste da parte dei client. Questa architettura consente al server di gestire efficacemente e simultaneamente numerose connessioni, riducendo al minimo l'overhead legato ai thread e garantendo un alto grado di efficienza nell'elaborazione delle richieste dei client. L'utilizzo di un selettore (**Selector**) nel contesto di **NIO** consente al server di monitorare più canali di comunicazione, determinando quali canali sono pronti per operazioni di lettura o scrittura, e facilita la gestione concorrente delle richieste. La **threadpool** svolge un ruolo cruciale nell'elaborazione delle richieste, consentendo al server di affrontare carichi di lavoro significativi in modo efficiente, specialmente quando le richieste richiedono tempi di elaborazione prolungati. Questa architettura è ampiamente adottata in applicazioni ad alta concorrenza, come i server web e le applicazioni di rete in tempo reale.

La gestione delle richieste passa quindi dalle seguenti fasi:

1. Il **main thread** accetta le nuove connessioni **TCP** in arrivo con **NIO.**
2. Appena un socket è pronto alla lettura il **main thread** legge il contenuto della richiesta e mette in coda un nuovo task (**RequestTask**) per la **threadpool** specificando la richiesta arrivata dal client.
3. La **thread pool** gestisce la richiesta appena possibile e ritorna la risposta nell'attachment della **SelectionKey.**
4. Il **main thread** appena un canale è pronto per la scrittura controlla se ci sono dati disponibili per il client (controllando l'attachment della **Key**) e li invia.

Una volta che il client stabilisce una connessione **TCP** con il server, questa viene mantenuta fino a quando il client stesso effettua il logout oppure se il server interrompe la comunicazione.

TCP PROTOCOL

Il protocollo di comunicazione **TCP** tra client e server avviene mediante lo scambio di particolari oggetti **JSON** ottenuti dalla serializzazione delle classi **TcpResponse** e **TcpRequest**.

Le richieste che il client invia al server utilizzano il seguente schema **JSON**:

request.json

```
{
      "command": string,
      "username": string,
      "data": string
}
```

Le risposte che il server invia al client sono invece nel seguente formato **JSON**:

response.json

```
{
     "code": string,
     "userGuess": {
         "letter": string,
         "guessStatus": string
      }[],
      "remainingAttempts": number,
      "wordTranslation": string,
      "stat": {
          "playedGames": number,
          "wonGamesPercentage": number,
          "avgAttemptsWonGames": number,
          "lastStreakWonGames": number,
          "bestStreakWonGames": number,
          "guessDistribution": {
                "attemptNumber": number,
                "percentage": number,
           },
      }
}
```

La risposta che il server invia al client non contiene in ogni caso tutti i tipi di dati riportati nello schema, ma verranno popolati a seconda della richiesta che il client inoltra al server.

Il campo più rilevante contenuto nella risposta del server è **code**. Questo infatti contiene una lista di codici necessari al client per interpretare correttamente la risposta ricevuta.

4.1.2 RMI

La classe **ServerMain** implementa l'interfaccia **ServerRMI** che permette al client di eseguire le seguenti azioni:

- **register()**: registrare un nuovo utente al sistema
- **subscribeClientToEvent()**: iscrive il client agli eventi che riguardano l'aggiornamento della classifica di gioco
- **unsubscribeClientToEvent()**: disiscrive un utente dagli aggiornamenti della classifica di gioco

Tutti i metodi sono marcati come **synchronized** per evitare che letture/scritture sulla struttura dati che mantiene la lista di utenti iscritti agli eventi creino inconsistenza dei dati.

4.1.3 RMI callback

RMI callback viene utilizzata dal server per comunicare ai vari client la classifica di gioco non appena ci sono dei cambiamenti nei primi 3 posti.
I client per poter essere avvisati devono necessariamente essersi iscritti agli eventi mediante la RMI **subscribeClientToEvent().**

4.1.4 Thread pool

La thread pool incaricata della gestione delle richieste **TCP** provenienti dai client è stata configurata attraverso l'utilizzo del costruttore **ThreadPoolExecutor**. La sua configurazione è molto simile a quella di una **CachedThreadPool** , con l'eccezione del numero massimo di thread, il quale è stato impostato al doppio dei core disponibili sulla macchina.

Questa configurazione è stata implementata al fine di mitigare il potenziale sovraccarico causato da un elevato numero di richieste da gestire. Tale sovraccarico potrebbe derivare da un'eccessiva creazione di thread, con conseguente degrado delle prestazioni causato da un pesante context switching.

Di seguito il codice del costruttore del **ThreadPoolExecutor**.

```
// Inizializza thread pool executor
int coreCount = Runtime.getRuntime().availableProcessors();
poolExecutor = new ThreadPoolExecutor(0, coreCount*2, 60L,
TimeUnit.SECONDS, new ArrayBlockingQueue<>(1000));
```

Si noti che è stata scelta la coda **ArrayBlockingQueue** per i task in quanto a differenza di una **CachedThreadPool** avendo impostato una dimensione massima per la pool è possibile avere molti task in attesa di essere eseguiti.

Tasks

I task che vengono aggiunti alla coda della threadpool sono delle istanze della classe **RequestTask** che implementa **Runnable.**
La classe **RequestTask** contiene tutti i metodi necessari a soddisfare uno dei seguenti comandi che il client può inviare al server:

```
public enum TCPCommandEnum {
   LOGIN,
   LOGOUT,
   PLAY_WORDLE,
   VERIFY_WORD,
   STAT,
   SHARE
}
```

Una volta che la richiesta è stata gestita, la risposta di tipo **TcpResponse** (sez. 4.1.1) viene aggiunta all attachment della **SelectionKey** passata nel costruttore di **RequestTask**.

4.1.5 Word extraction

L'estrazione della nuova parola viene fatta in modo parallelo rispetto al main thread con il task **WordExtractorTask** che viene invocato periodicamente a seconda del parametro di configurazione del server ***app.wordle.word.time.minutes*** mediante il service **ScheduledExecutorService.**

Durante le operazioni di estrazione e traduzione della nuova parola, viene preso il **lock** sulla struttura dati che mantiene la parola attuale in modo tale che nessun altro thread acceda alla parola durante il suo aggiornamento (vedi sez. 7.). Questo evita che i giocatori continuino ad utilizzare una parola che è in corso di aggiornamento.

Traduzione parola

Una volta estratta la parola dal dizionario questa viene tradotta grazie alla API messa a disposizione da **mymemory.translated.net**.

La richiesta di traduzione di una parola viene fatta mediante una richiesta **HTTP** di tipo **GET** al seguente indirizzo: **api.mymemory.translated.net/ge**t con i seguenti query param:

- q=<parola da tradurre>
- langpair=<lingua partenza|lingua destinazione>

Di seguito viene riportata un esempio di **cURL** alle API di tradizione:

```
curl 'https://api.mymemory.translated.net/get?q=house&langpair=en|it'
```

NOTA:
In caso di problemi di connessione ad internet da parte del server, la traduzione della parola potrebbe richiedere un tempo molto elevato e per questo motivo **WordExtractorTask** potrebbe mantenere la lock troppo a lungo sulla parola attuale. Per evitarlo è stato aggiunto un timeout di 5 secondi per la chiamata **GET** entro il quale le API devono ritornare una risposta.
Nel caso in cui si raggiunga il tempo massimo dei 5 secondi la parola non viene tradotta, ma per evitare inconsistenza si ritorna la parola non tradotta.

4.1.6 SIGINT & SIGTERM

Per garantire una corretta chiusura del server e salvataggio dei dati in qualsiasi occasione è stata implementata la classe **ServerShutdownHook** (estende **Thread**) il cui metodo **run()** viene lanciato non appena il processo riceve un segnale di **SIGINT** o **SIGTERM**.
Questo e' stato possibile grazie alla funzione **addShutdownHook()** che permette di agganciare un qualsiasi thread alla ricezione di segnali di interruzione:

```
// Hook per SIGINT e SIGTERM
Runtime.getRuntime().addShutdownHook(new ServerShutdownHook());
```

Di seguito viene riportato il dettaglio del metodo **run()** della classe **ServerShutdownHook.**

```
@Override
public void run() {
    logger.info("Terminazione Wordle server...");

    // Richiesta di terminazione graduale del thread pool
    ServerMain.poolExecutor.shutdown();
    try {
        // Attendo che la threadpool sia terminata per un massimo di 10 secondi
        if (ServerMain.poolExecutor.awaitTermination(10, TimeUnit.SECONDS)) {
               logger.debug("Thread pool terminata correttamente");
        }
    } catch (InterruptedException ignore) {}

    // Interrompo word update
    ServerMain.wordUpdateExecutor.shutdown();
    // Salvo utenti su file
    this.userService.saveUsers();
    // Salvo stato del gioco su file
    this.wordleGameService.saveState();
    // Chiudo socket multicast
    ServerMain.multicastSocket.close();
    // Chiudo socket channel
    try {ServerMain.socketChannel.close();} catch (IOException ignore) {}
}
```

```
});
```

4.2 Client

La classe principale del client è **ClientMain** situata nella cartella src/main/java/client.
Il client opera in 3 modalità:

- **Guest mode** -> utente non ancora loggato
- **User mode** -> utente loggato
- **Game mode** -> modalità gioco

Di seguito viene riportato il codice che cicla sulle varie modalità di funzionamento del client:

```
public void run() {

   while (true) {
      switch (mode) {

         case GUEST_MODE:
            this.guestMode();
            break;

         case USER_MODE:
            this.userMode();
            break;

         case GAME_MODE:
            this.gameMode();
            break;
      }
   }
}
```

4.2.1 Guest mode

Questa è la modalità con cui viene avviato il Wordle client. In questa fase l'utente non è ancora loggato e le azioni disponibili sono limitate.

Fig. Screenshot modalità ospite

I comandi disponibili per l'utente sono limitati e comprendono le operazioni di ***login***, ***registrazione***, ***help*** e ***quit***.

4.2.2 User mode

Dopo essersi loggato l'utente passa dalla **guest mode** alla **user mode**. In questa fase il menu principale proposto comprende tutti i comandi disponibili nell'applicazione:

Fig. Screenshot modalità utente

Play

Questo comando permette al giocatore di avviare una nuova partita a Wordle.
Nel caso di successo il client passerà alla game mode (**sez. 4.2.3**), mentre in caso di fallimento (parola già indovinata, errore del server, ecc..) verrà presentato nuovamente il menu.

Share

Comando per richiedere al server di condividere l'ultima partita giocata sul gruppo multicast. Nel caso in cui l'utente non abbia partite giocate, il comando non avrà nessun effetto.

Social

Mostra tutte le condivisioni dei client ricevute mediante gruppo multicast (sez. 4.3).

Fig. Risultati condivisi da utenti

Stat

Richiede al server le statistiche del giocatore e mostra a video il seguente contenuto:

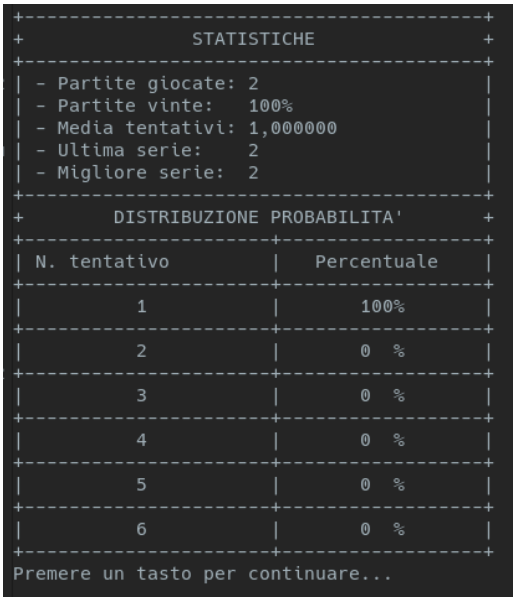

Fig. Statistiche utente

La distribuzione di probabilità mostra come sono distribuiti i vari tentativi fatti dall'utente nelle partite vinte.

Rank

Mostra la classifica di gioco attuale, sulla base del punteggio ottenuto con la seguente formula:

*score = (partite vinte) * (media tentativi)*

La classifica viene inviata dal server verso i client mediante **RMI callback** (sez. 4.1.3).

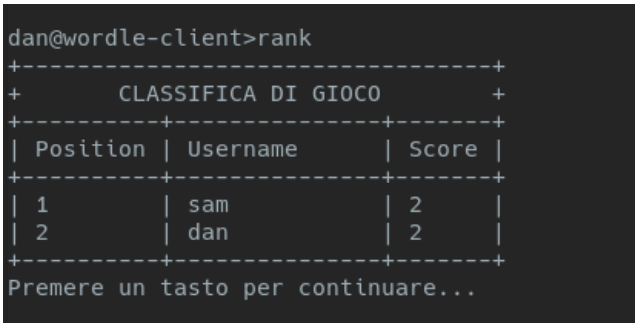

Fig. Classifica di gioco

4.2.3 Game mode

La modalità di gioco permette all'utente di avere un'esperienza di gioco "immersiva" senza distrazioni e senza la necessità di fornire comandi ripetitivi per mandare un nuovo tentativo di guess al server.

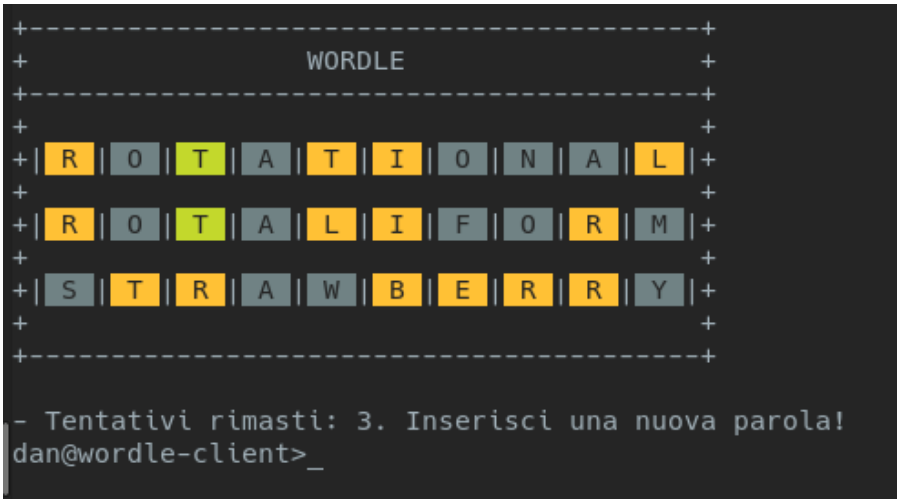

Fig. Screenshot modalità di gioco

Ad ogni tentativo inserito la CLI mostra i suggerimenti ricevuti dal server per i tentativi inviati fino ad adesso. Viene anche presentato il numero di tentativi rimanenti alla conclusione del gioco.

Una volta terminato il gioco, il client presenta un riepilogo delle partite giocate fino a quel momento.

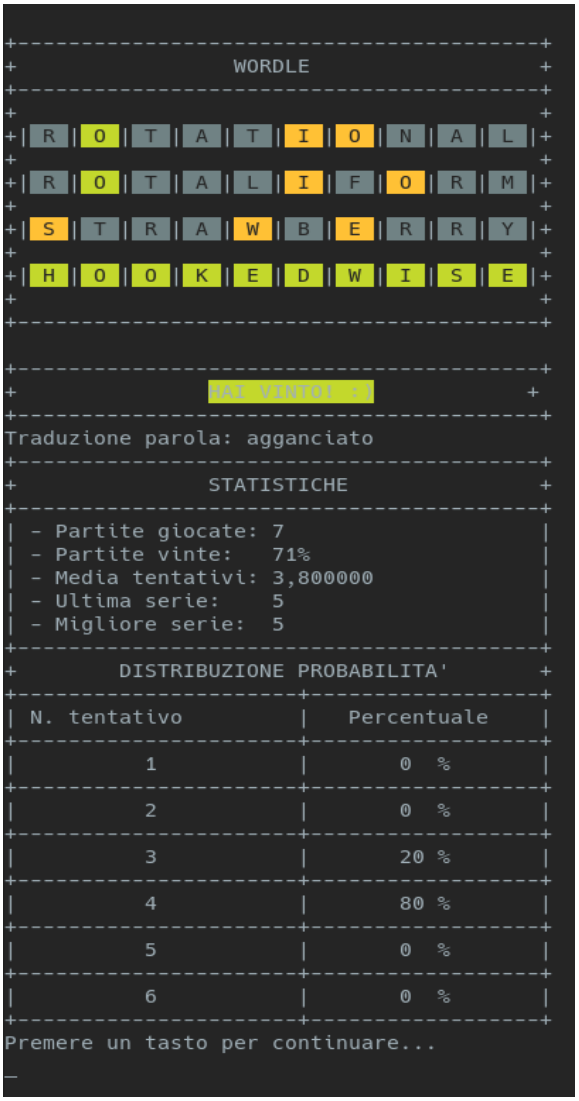

Fig. schermata vittoria partita

4.3 Multicast daemon

Il gruppo **multicast** viene utilizzato dal server per condividere a tutti i client partecipanti il risultato dell'ultima partita completata del client di cui ne fa richiesta.

Ogni client dispone di un thread demone sempre attivo in attesa di nuovi messaggi in arrivo sul gruppo multicast.
All'arrivo di un nuovo messaggio il demone aggiunge la condivisione ricevuta in una struttura dati utilizzata poi su richiesta dell'utente per visualizzare tutte le condivisioni ricevute.
Il demone viene terminato automaticamente alla chiusura del server.

La decisione di implementare un thread demone è stata presa tenendo in considerazione che, essendo una funzionalità secondaria, può essere eseguita in modo meno prioritario dal sistema.
Il demone viene chiuso automaticamente dalla JVM alla terminazione del client.

4.3 SIGINT & SIGTERM

Come nel caso del server, anche per il client è stata implementata la classe **ClientShutdownHook** (estende **Thread**) che esegue la corretta sequenza di azioni per la terminazione del client.
Di seguito il dettaglio del metodo **run()** della classe:

```
@Override
public void run() {

    logger.debug("Shutdown Wordle client...");

    try {
        // Disiscrivo client da eventi del server
        if (this.client.username != null) {
               this.serverRMI.unsubscribeClientFromEvent(this.client.username);
               // Effetto il logout
               this.client.logout();
        }
    } catch (RemoteException e) {
        logger.error("Errore chiamata RMI unsubscribeClientFromEvent()");
    }

    try {
        // Chiudo il socket TCP con il server
        this.socket.close();
    } catch (IOException e) {
        logger.error("Errore durante chiusura socket TCP");
    }

    System.out.println("Grazie per aver usato Wordle client! Torna presto!");
}
```

5. Thread attivi

In questa breve sezione viene riportato uno schema riassuntivo dei thread attivi in ogni momento sul server e sul client.

5.1 Server

I thread attivi sul server sono i seguenti:

- **Main thread:** esegue il multiplexing dei canali
- **Threadpool:** per la gestione delle richieste
- **Word update thread:** per l'estrazione periodica delle parole

5.2 Client

I principali thread attivi sul client sono:

- **Main thread:** rimane in attesa di comandi dell'utente
- **Multicast daemon:** per il salvataggio degli eventi in arrivo sul gruppo multicast

6. File di configurazione

I file di configurazione sono **necessari** per il corretto funzionamento sia del server che del client e possono essere passati alle applicazioni usando la variabile di ambiente **WORDLE_CONFIG**.
Il formato dei file di configurazione e' il seguente:

server.config

```
# TCP configuration
app.tcp.port=5783

# RMI configuration
app.rmi.port=9876

# Multicast configuration
app.multicast.ip=226.226.226.226
app.multicast.port=4000

# Game settings
app.wordle.word.time.minutes=2
```

client.config

```
# TCP configuration
app.tcp.port=5783
app.tcp.ip=127.0.0.1
# socket read/write timeout(ms)
app.tcp.timeout=5000

# Multicast configuration
app.multicast.ip=226.226.226.226
app.multicast.port=4000

# RMI configuration
app.rmi.port=9876
```

I precedenti file di configurazione vengono letti dal server e dal client utilizzando una classe helper sviluppata specificatamente per questo scopo (**sez.11**).

NOTA
I file sopra elencati sono stati inclusi anche nel progetto per avere dei file di esempio base.
Attenzione! Non è possibile avviare né il client né il server senza i file di configurazione e tutte le proprietà sopra elencate sono obbligatorie!

7. Persistenza

I dati che riguardano gli utenti e le parole estratte dal server sono persistenti ai riavvii del server.

Ad ogni avvio il server controlla se sono presenti file da ripristinare nella cartella **data/**, altrimenti inizializza il sistema per il primo avvio.
I dati vengono salvati nel formato **JSON** nei seguenti file:

- **users.json**
- **wordle.json**

Di seguito sono riportati degli esempi delle strutture dati dei file:

wordle.json

```
{
 "actualWord": "abscission",
 "extractedAt": "Oct 12, 2023, 15:16:07",
 "gameNumber": 37
}
```

users.json

```
{
 "username": "dan",
 "password": "KtLuWi+q2SqowX+ScoapeA\u003d\u003d",
 "salt": "Rw+OP7sgyf8uAqFtTlhihA\u003d\u003d",
 "lastStreak": 2,
 "bestStreak": 2
},
```

I file vengono salvati alla chiusura del server, ovvero quando l'applicazione termina.
Questo risulta vantaggioso perché le operazioni su disco vengono fatte solamente all'avvio e allo spegnimento del server riducendo l'overhead dovuto a letture/scritture frequenti.
Per semplificare e unificare il codice che effettuasse la lettura dei JSON e' stata implementata la classe helper **JsonService**.
I metodi più rilevanti in questa classe sono i seguenti:

```
public static Object readJson(String path, Type type);

public static <T> void writeJson(String path, T object);
```

Il metodo **readJson()** legge il contenuto del file **JSON** specificato dall'argomento *path* ed effettua il parsing utilizzando il tipo fornito dall'argomento *type*.

Il metodo **writeJson()** consente la scrittura di un oggetto generico di tipo T nel file specificato dal primo argomento. Inoltre, questo metodo gestirà la creazione dell'intera struttura delle directory per raggiungere il file nel caso in cui non siano già presenti tutte le cartelle necessarie.

8. Strutture dati condivise

Le principali strutture dati condivise tra i thread del server sono la lista di utenti registrati con le loro relative partite e le informazioni che riguardano il corrente stato del gioco.

Tutte le strutture sono accessibili mediante due classi helper:

- **UserService**
- **WordleGameService**

Queste classi permettono l'accesso sicuro alla strutture dati condivise fornendo dei metodi **synchronized** che evitano le race conditions tra i diversi thread oltre a fornire tutta una serie di funzioni utili per effettuare operazioni più complesse sugli utenti e sul gioco.
L'accesso alla parola attualmente estratta dal server viene reso sicuro grazie ad una **ReentrantLock** che garantisce che nessun thread acceda alla parola nel momento in cui viene aggiornata dal task periodico (vedere **sez. 4.1.4**).

Si consiglia una visione del codice sorgente per avere informazioni più dettagliate su questo aspetto.

9. Sicurezza dati

Il maggior rischio per la sicurezza dei dati degli utenti registrati a **Wordle** riguarda le credenziali di accesso.
Questi dati sono tra le informazioni che il sistema deve persistere tra i riavvii e conseguentemente si ha la necessità di salvarli su file (users.json, vedi **sez. 6.**).

In questa implementazione le password degli utenti non vengono memorizzate direttamente in chiaro, ma piuttosto vengono elaborate attraverso una funzione hash utilizzando l'algoritmo **PBKDF2**.
Questo processo coinvolge anche l'uso di un valore casuale noto come "**salt**", il quale è diverso per ciascun utente. L'utilizzo di un salt unico per ogni utente migliora la sicurezza dell'hashing, poiché anche se due utenti dovessero avere la stessa password, i loro hash saranno differenti a causa del salt diverso. Questo approccio è un buon metodo per proteggere le password dagli attacchi brute force.

Di seguito viene mostrato un esempio di cosa viene salvato su file dopo la registrazione di un utente.


```
{
 "username": "dan",
 "password": "q8zNom+TIFUSlpcnKefFtQ\u003d\u003d",
 "salt": "ioabztJeLSjXlGJuNS3fJg\u003d\u003d"
}
```


10. Log

È stata dedicata particolare attenzione alla registrazione dei log, essenziali soprattutto sul server per ottenere una visione completa degli eventi in corso. A tal fine, è stata sviluppata una classe di supporto denominata **WordleLogger**, che consente di avere i seguenti livelli di log:

- Info
- Warn
- Debug
- Error
- Success

In base alla gravità del log, viene utilizzato un colore diverso per la formattazione. Inoltre, ogni log contiene le seguenti informazioni:

- Severità del log
- Thread associato
- Classe da cui proviene il log
- Timestamp

Questo approccio è estremamente utile per il monitoraggio, la risoluzione dei problemi e il tracciamento delle attività all'interno del server.

Nota: Per abilitare i log di debug e' necessario impostare la variabile di ambiente WORDLE_DEBUG=true (vedi sez. 3.2)

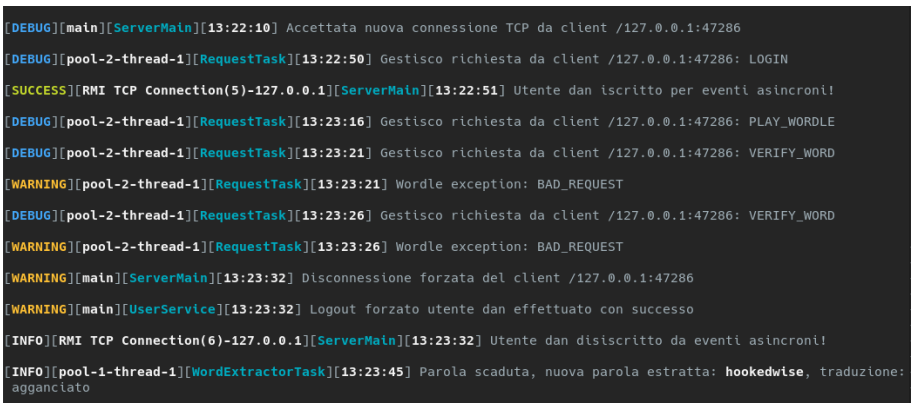

Fig. Screenshot server log

11. Utils

Per facilitare lo sviluppo e la manutenzione del codice sono state create alcune classi di supporto utili sia all'applicazione server sia all'applicazione client.

Le principali classi di supporto sono le seguenti:

- **ConfigReader**
- **WordleLogger**

Come anticipato nella sez.10 la classe **WordleLogger** viene utilizzata come helper delle funzioni di logging.

La classe **ConfigReader** invece contiene i seguenti metodi:

```
public static Properties readConfig();

public static String readProperty(Properties properties, String
propertyName) throws NoSuchFieldException
```

Il metodo **readConfig()** permette di leggere il file di configurazione che si trova nella posizione specificata dalla variabile di ambiente **WORDLE_CONFIG.**
Nel caso in cui la variabile d'ambiente non sia stata correttamente passata, l'intero programma termina (fatal error). Il metodo ritorna un oggetto **Properties** in caso di successo.

Il metodo **readProperty()** accetta come argomenti un insieme di **Properties** lette grazie al metodo precedente ed effettua il parsing del campo con nome **propertyName**. Nel caso in cui il campo cercato non sia presente nelle **Properties** viene lanciata eccezione.
La condivisione di questi helper sia per il client che per il server ha permesso di ridurre il numero di righe di codice duplicate.

12. Implementazione Wordle

Il gioco Wordle implementato in questo progetto segue le regole principali fornite nelle specifiche del progetto ma sono state effettuate scelte personali per alcune parti del gioco.
In questa sezione sono elencate le scelte più importanti lasciate alla libera scelta.

12.1 Play Wordle

Come mostrato negli screenshot della sez. 4.2.2 non è presente il comando specifico per richiedere al server di iniziare il gioco con la parola attuale.
In questa implementazione è stato scelto di inviare il comando di richiesta inizio gioco ogni volta che l'utente invia il comando ***play***. Questo permette di migliorare l'esperienza utente e ridurre la complessità per l'avvio di una nuova partita.

12.2 Guess word

Come precedentemente specificato nella **sez. 4.1.5**, durante il processo di estrazione di una parola, il thread acquisisce il lock sulla struttura dati che contiene la parola attuale. Di conseguenza, tutte le richieste da parte dei client per avviare una nuova partita vengono messe in attesa fino a quando la nuova parola non è stata estratta con successo.

Nonostante ciò prevenga che nuovi client inizino a giocare con una parola non più valida, questo meccanismo non impedisce che una nuova parola venga estratta immediatamente dopo che uno o più client abbiano ricevuto l'autorizzazione per iniziare a giocare. In tale situazione, quando un client invia una "guessed word" (parola indovinata) per la parola attuale, il server risponderà con un codice di errore, invitando il client a riavviare la partita. In questo particolare caso, il server considera la partita come annullata e di conseguenza i risultati non verranno considerati nelle statistiche del giocatore.

12.3 Social share

Le partite che i client decidono di condividere con gli altri giocatori attraverso il gruppo multicast non vengono conservate dal server in alcun modo. Di conseguenza, ad ogni riavvio del server, le condivisioni degli utenti vengono azzerate. Questa situazione si applica anche nel caso in cui un utente effettua il logout e, successivamente, il login.