Wordle - Progetto di laboratorio di reti

Yuri Caprini

November 2023

Contents

1 Manuale utente

La gestione dell'intero progetto avviene tramite il tool *Apache Maven*. Prima di procedere alla compilazione è necessario dunque assicurarsi che tale strumento sia installato nel proprio ambiente di lavoro. Per farlo si prega di consultare la pagina ufficiale `https://maven.apache.org/`. Assicurarsi anche che una versione di *Java* uguale o superiore alla 8 sia installata sul proprio sistema.

1.1 Compilazione ed avvio

Per compilare ed eseguire il progetto procedere con i passi seguenti:

- Decomprimere il file `progetto.zip`
- Da terminale accedere alla cartella `progetto/wordle`, risultante dalla decompressione, in modo tale da renderla la propria cartella di lavoro.

- Una volta all'interno della cartella, eseguire il comando `mvn package`
- Adesso da un secondo terminale accedere in modo analogo alla cartella `progetto/wordle/wordleserver/target/` ed avviare il server eseguendo il comando:

  `java -jar server-executable-jar-with-dependencies.jar`
- Tornare ora al primo terminale, accedere a `wordleclient/target/` ed avviare il client eseguendo il comando:

  `java -jar client-executable-jar-with-dependencies.jar`

È possibile saltare la fase di compilazione facendo uso diretto, con le stesse modalità, dei jar eseguibili forniti all'interno della cartella `progetto`.

È anche possibile passare come argomento da terminale, ad ognuno dei due jar eseguibili, il path di un file json di configurazione in modo tale che l'eseguibile legga i parametri da quel file invece che da quello memorizzato all'interno del jar stesso. Come template sono disponibili nella cartella `progetto` i file `server_config_template.json` e `client_config_template.json`.

1.2 Utilizzo del client

Il client interagisce con l'utente tramite CLI (Command Line Interface). È possibile dunque interagire con quello appena avviato sul primo terminale digitando da esso uno o più comandi fra quelli disponibili:

- `register user Passw0rd` : registra un nuovo utente a *Wordle* stabilendo come credenziali di accesso quelle fornite. Il nome utente deve essere compreso tra i 4 e 10 caratteri e non deve contenere spazi. La password deve essere compresa tra gli 8 e 16 caratteri, non deve contenere spazi, ma deve contenere almeno un carattere maiuscolo e almeno un numero.
- `login user Passw0rd` : consente all'utente di accedere a *Wordle* inserendo le credenziali scelte in fase di registrazione. Uno stesso utente può effettuare un accesso contemporaneo da dispositivi differenti.
- `logout` : permette all'utente di uscire dalla propria sessione e tornare alla fase di accesso. Il logout non comporta la perdita della partita in corso, che può essere ripresa effettuando un nuovo login.
- `playwordle` : consente all'utente di iniziare una nuova partita se e solo se una nuova parola da indovinare è stata estratta. In tal caso, se una partita è in corso, l'esecuzione del comando provoca la perdita automatica della partita in corso e l'inizio di una nuova partita. Se invece nessuna nuova parola da indovinare è stata estratta e una partita è in corso, visualizza gli indizi di quest'ultima in modo che l'utente possa continuarla.
- `sendword yourguess!` : consente all'utente di effettuare un tentativo nel cercare di indovinare la parola segreta nella partita in corso.

- `showmestats` : mostra all'utente le proprie statistiche di gioco.
- `showmeranking` : mostra all'utente la classifica dei giocatori.
- `share` : permette all'utente di condividere con gli altri utenti connessi i risultati dell'ultimo gioco terminato nella sessione.
- `showmesharing` : mostra all'utente le condivisioni dei risultati di gioco da parte degli altri utenti.
- `showmetop3` : mostra l'aggiornamento (avvenuto tramite notica) delle prime 3 posizioni della classifica giocatori.
- `help` : illustra all'utente la lista dei comandi disponibili.
- `quit` : termina il client.

2 Organizzazione e struttura del progetto

Il progetto è organizzato in tre moduli:

- `wordleprotocol`: contiene le interfacce che modellano il protocollo di comunicazione client-server e le loro implementazioni. Fornisce anche le strutture dati utilizzate dal client e dal server per inviare/ricevere richieste/risposte.
- `wordleserver`: contiene le interfacce e le classi che realizzano il server organizzate in package secondo i principi della *Clean Architecture* (di più nella sezione dedicata).
- `wordleclient`: contiene le (poche) classi che realizzano un *thin client*.

2.1 Modulo protocollo

Secondo specifica sono previsti tre modalità di comunicazione tra client e server:

1. Richieste/risposte inviate/ricevute tramite una connessione TCP:
   sono modellate dalle interfacce `WordleRequest` e `WordleResponse` le cui factory vincolano con esattezza la loro costruzione. Sono implementate come richieste e risposte HTTP minimali costruite, codificate e decodificate tramite classi custom, senza l'uso di librerie esterne. Il payload di tali richieste/risposte, quando presente, consiste in DTO (Data Transfer Object) serializzati in formato json, le cui classi risiedono nel package `dtos`. Poichè l'autenticazione avviene tramite token JWT è presente anche una classe wrapper che vincola le funzionalità di tali token alle uniche necessarie ai fini del progetto. Si trova nel package `auth`.
2. Dati inviati tramite pacchetti UDP:
   Consistono nella semplice serializzazione di alcuni dei DTO descritti sopra.

3. Dati inviati tramite RMI (Remote Method Invocation):
   le interfacce che modellano questo tipo di comunicazione si trovano nel package `remoteinterfaces`.

Manca un ultimo package da nominare, chiamato `ioutils`, contenente le interfacce che modellano le code di input e di output utilizzate dal client e dal server per l'invio e la ricezione dei dati.

Si noti che `wordleclient` e `wordleserver` dipendono solo dalle interfacce di questo modulo, il che rende il cambio delle rispettive implementazioni a loro trasparente. Una visione complessiva dei package, delle interfacce e delle classi che compongono il modulo e delle loro dipendenze è data nella vista in figura 1.

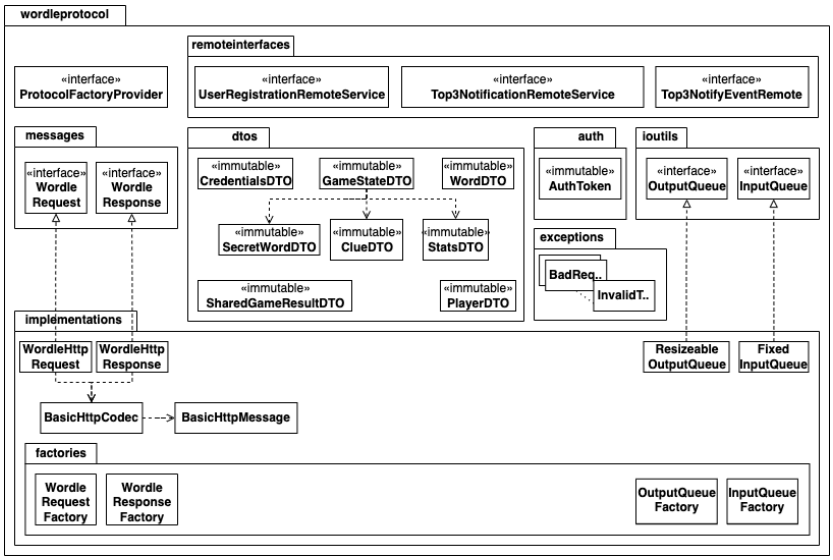

Figure 1: Vista del modulo protocollo

2.2 Modulo server

Il modulo server segue i principi di design della *Clean Architecture*, concetto sviluppato da Robert C. Martin (noto anche come "Uncle Bob"), volto a mettere al centro della progettazione del software il dominio del problema in modo che esso sia indipendente da framework, dataprovider, librerie esterne... Esse sono considerate come "dettagli".

Per fare questo si deve immaginare il server strutturato "a cipolla", il cui nucleo rappresenta il livello di astrazione più alto che modella il dominio del problema e le logiche di business, mentre gli strati esterni rappresentano i livelli più concreti dove risiedono componenti e logiche che realizzano effettivamente l'applicazione. Dunque, andando dagli strati esterni verso quelli interni avviene un processo di *astrazione*, mentre viceversa un processo di *concretizzazione*.

Regola fondamentale è che i componenti degli strati interni non dipendano *mai* da quelli esterni, cosi che l'applicazione sia flessibile al cambio di framework e tecnologie (una su tutte il modo con cui si recuperano o memorizzano i dati). Il risultato è una struttura flessibile, manutenibile, testabile e una organizzazione del codice che "grida" (si parla di anche di *screaming architecture*) le funzionalità e il significato di ogni componente. Vediamo se è vero, dando uno sguardo alla vista dei package che compongono `wordleserver` in figura 2.

Come evidenziato dalla colorazione, il modulo si compone di cinque package principali che rappresentano gli "anelli della cipolla". Descriviamoli brevemente partendo dal più interno al più esterno:

- `circle01entities`: contiene le classi rappresentanti le entità del dominio *Wordle*. Esse costituiscono gli elementi fondamentali su cui sono costruite le regole di business. È qui che si definiscono i vincoli che caratterizzano una `Password` o uno `Username`, il criterio con cui si calcola lo `Score` di un giocatore, le regole di gioco e cosi via.

- `circle02usecases`: contiene le classi rappresentanti i casi d'uso. Essi interagiscono con le entità per realizzare le regole di business specifiche dell'applicazione. È qui che si decidono le operazioni da eseguire e come rispondere a seguito di una richiesta specifica di un attore del sistema. Si noti come esiste quasi un rapporto 1:1 tra le classi definite in questo package e i casi d'uso del diagramma in figura 3 realizzato preliminarmente in fase di analisi del dominio.

- `circle03adapters`: contiene le classi che giocano il ruolo di adattatori convertendo i dati tra il formato più conveniente per i casi d'uso e quello più conveniente per i componenti degli anelli esterni.

- `circle04frameworks`: contiene i componenti dei framework che si occupano concretamente della ricezione/invio delle richieste/risposte, e dell' attivazione, gestione e chiusura dei thread necessari all'esecuzione.

- `circle05configurations`: contiene le classi che si occupano della configurazione e del collegamento delle componenti dell'intero sistema e del suo bootstrap.

Un ultimo aspetto da notare è che nella vista in figura 2, come anticipato, non esiste nessuna freccia di dipendenza che parte da un componente di un anello interno e arriva ad un componente di un anello più esterno. Questo spesso è ottenuto tramite una *inversione della dipendenza* di un componente interno nei confronti di uno esterno interponendo tra i due una interfaccia da cui il primo dipende e che il secondo implementa.

2.3 Modulo client

Questo modulo è il più modesto dei tre, dato che il client presenta funzionalità ridotte. La classe `ClientMain` si occupa del collegamento dei componenti, del

caricamento della `ClientConfiguration` e del bootstrap dell'applicazione. La classe `Client` presenta la logica che gestisce il loop di interazione con l'utente e implementa l'interfaccia remota `Top3NotifyEventRemote` realizzando il paradigma di comunicazione RMI callback con il quale vengono ricevuti gli aggiornamenti delle prime 3 posizioni della classifica giocatori. Infine la classe mantiene un riferimento ad uno `SharedResultsListener` che si occupa di ricevere i risultati di gioco condivisi dagli altri giocatori. La vista di `wordleclient` è in figura 4.

3 Dinamica del server

All'avvio il server manda in esecuzione cinque `Service`[1] distinti e nel seguente ordine:

1. `PersistenceService`: all'avvio si occupa di caricare in memoria le entità lette da un file json ricostruendo lo stato del sistema, poi si sospende fino al segnale di spegnimento del server, quando si risveglia un' ultima volta per memorizzare nuovamente lo stato del sistema. Sospensione e risveglio avvengono tramite `wait()` e `notify()`.

2. `SecretWordRefresherService`: si occupa di aggiornare periodicamente la parola segreta che l'utente deve indovinare, dandogli di fatto la possibilità di giocare nuove partite. Ad ogni aggiornamento segue un periodo di sospensione che avviene tramite una semplice `sleep()`. Il servizio è interrotto questa volta tramite `interrupt()`.

3. `RemoteExposerService`: all'avvio esporta su `Registry` gli oggetti remoti necessari per effettuare la registrazione utente e la condivisione dei risultati di gioco per mezzo del paradigma di comunicazione RMI e callback RMI. Dopodichè si sospende fino al segnale di spegnimento quando si riattiva un ultima volta per effettuare l'unexport dei medesimi oggetti. Anche qui analogamente al primo servizio, sospensione e riattivazione avvengono tramite `wait()` e `notify()`.

4. `DispatcherService`: effettua costantemente un ciclo di selezione dei canali di comunicazione associati alle connessioni client, scegliendo quelli che sono pronti ad effettuare operazioni di lettura o scrittura per le quali avevano manifestato un interesse. Per entrare a far parte del ciclo di selezione i canali vengono registrati ad un `Selector` interno al dispatcher, assieme ai loro `ChannelHandler`, tramite una `RegistrationFacade`. Quando un canale è selezionato, in quanto pronto ad effettuare una operazione di lettura o scrittura, il `ChannelHandler`[2] associato è mandato in esecuzione su un thread dedicato facente parte di un pool[3]. Al termine della sua esecuzione, il gestore del canale è accodato in una `BlockingQueue` che agisce

[1] L'interfaccia `Service` estende `Callable` ed ogni `Service` è eseguito da un thread distinto.
[2] Anche `ChannelHandler` estende `Callable`.
[3] Il numero di thread del pool è fissato a `availableProcessors() * 2` per limitare fenomeni di *over-subscription* che possono degradare le performance. Altri framework, tra cui *Netty*, adoperano questa scelta.

da punto di sincronizzazione da cui il thread dispatcher lo preleverà per registrarlo nuovamente al selettore nel caso in cui abbia altre operazioni di I/O da effettuare o per rimuoverlo semplicemente dalla coda nel caso in cui la sua esecuzione logica sia terminata.

5. `ListenerService`: effettua costantemente un ciclo di ascolto delle connessioni entranti, le accetta, recupera i canali di comunicazione client e li registra al dispatcher assieme ai loro gestori per mezzo della citata `RegistrationFacade`. Anche qui il punto di sincronizzazione tra listener e dispatcher è la `BlockingQueue` nominata al punto precedente.

In fase di spegnimento del server (invocato tramite `Ctrl+C`) l'ordine di terminazione dei servizi è inverso a quello d'avvio cosi che lo stato del server persistito sia consistente con le operazioni effettuate dai client.

Un qualsiasi errore critico nell'esecuzione di uno qualsiasi di questi servizi è gestito catturando l'eccezione correlata nel rispettivo metodo `done()` e porta alla terminazione *graceful* del server, per quanto possibile. Invece, errori nella comunicazione con i client, per esempio durante una operazione di I/O, comportano esclusivamente la terminazione del gestore canale associato e la disconnessione del client.

4 Dinamica del client

Il ciclo di vita del client è assai più semplice di quello del server: all'avvio viene mandato in esecuzione un thread che esegue un ciclo di interpretazione dei comandi che l'utente digita su terminale. Un comando può corrispondere all'invio di un'unica richiesta al server di cui si attende immediatamente risposta (una eccezione è il comando `showmeranking` di cui parleremo nella sezione successiva) oppure può essere usato per mostrare il contenuto di dati mantenuti localmente. Dopo il successo di un comando di `login` si registra tramite paradigma RMI la callback per ricevere le notifiche di aggiornamento delle prime 3 posizioni della classifica e si avvia un thread di ascolto delle condivisioni dei risultati di gioco notificate a un gruppo multicast. Il comando `quit` pone termine al thread di ascolto e al loop di interazione.

5 Le strutture dati

Lo stato del server è costituito da due strutture dati principali:

- `RegisteredUsers`: contiene tutte e sole le informazioni riguardanti gli utenti registrati a *Wordle*. È implementata tramite una `ConcurrentHashMap` per evitare efficientemente fenomeni di inconsistenza dovuti alla concorrenza (per esempio senza di essa potrebbe accadere che due utenti con lo stesso nome siano in grado di registrarsi). I valori di questa mappa sono le entità rappresentate dalla classe `User` che fornisce ai casi d'uso i metodi necessari per realizzare le logiche di business dell'applicazione: iniziare

una partita, tentare di indovinare una parola, recuperare le informazioni riguardanti statistiche e risultati di gioco. Poichè è stato reso possibile ad uno stesso utente di effettuare l'accesso da terminali (e quindi dispositivi) differenti, per evitare fenomeni di inconsistenza (seppur poco probabili), tali metodi sono etichettati come `synchronized`.

- `Ranking`: rappresenta la classifica dei giocatori ordinata per punteggio. È implementata tramite una `TreeMap` per permettere un ordinamento efficiente e consistente ad ogni aggiornamento. Anche qui per le stesse ragioni precedenti i metodi di accesso alla struttura sono etichettati come `synchronized`. Cogliamo inoltre l'occasione per chiarire quanto accenato nella sezione 4 riguardo al metodo `showmeranking`: la classifica non è inviata al client per intero in un'unica risposta, ma per pagine, tramite un ciclo di richieste e risposte, implementando la tecnica del *cursor paging*. Questo consente al server, nel caso di un gran numero di richieste multiple concorrenti, di "respirare" tra l'invio di una pagina e l'altra, sia in termini di tempi di elaborazione che di memoria. Eventuali fenomeni di duplicazione dei dati dovuti all'uso della paginazione in ambiente concorrente sono corretti lato client.

Lato client nominiamo solo:

- `volatile PlayerDTO[] top3`: la variabile che ha come valore il riferimento all'array contenente le prime 3 posizioni della classifica. Il valore di tale variabile è aggiornato atomicamente all'arrivo di ogni notifica inviata dal server tramite RMI callaback.

- `sharedResults`: una `BlockingQueue` di dimensione fissata che si comporta come una coda circolare, rimuovendo le notifiche dei risultati di gioco più vecchie, nel caso in cui essa risulti piena all'arrivo di una nuova notifica.

6 Note a margine

Elenchiamo qui alcune note marginali, che non hanno trovato posto nella trattazione, ma che possono tornare utili al lettore:

- Il progetto non copre aspetti legati alla sicurezza, concentrandosi invece sui concetti di robustezza, manutenibilità e chiarezza.

- La CLI del client fa uso della libreria *Jansi* per mostrare gli indizi di gioco come parole dalle lettere colorate, cosi come avviene nel gioco reale. Unica differenza è che per chiarezza, il colore grigio è stato sostituito in rosso. La libreria utilizzata dovrebbe avere una alta portabilità e funzionare sulla maggior parte dei terminali.

- La versione di *Java* utilizzata per testare il codice è la `1.8.0_202`, quella di *Apache Maven* è la `3.8.5` e il sistema operativo è *MacOS X* versione `10.16`.

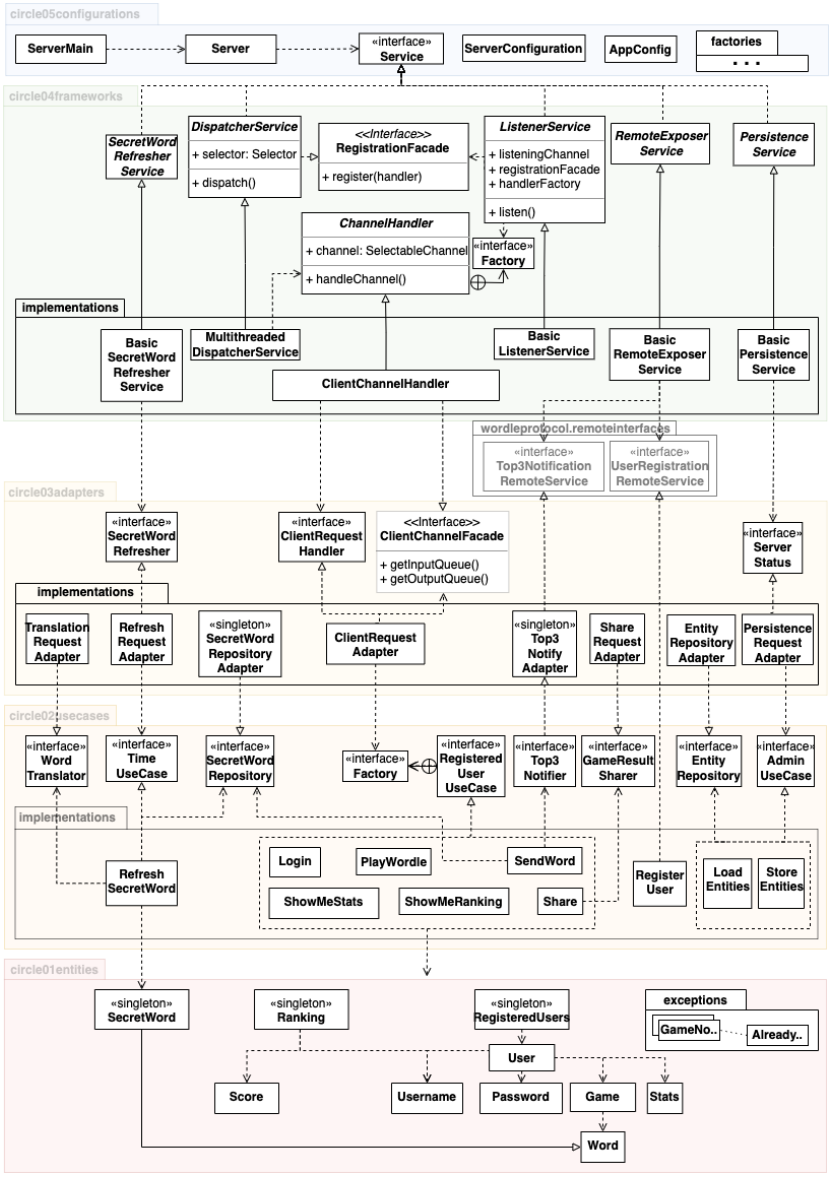

Figure 2: Vista del modulo server

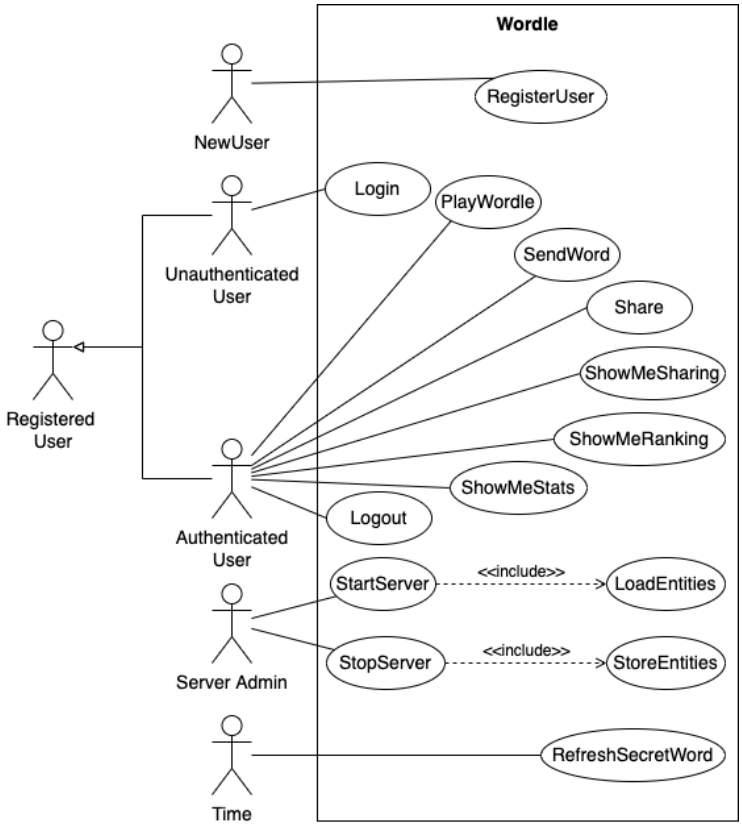

Figure 3: Diagramma dei casi d'uso

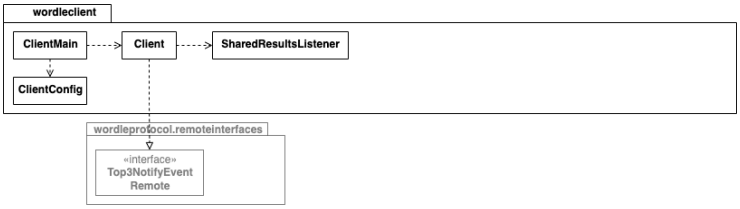

Figure 4: Vista del modulo client