UNIVERSITÀ DI PISA

==========

LABORATORIO DI RETI DI CALCOLATORI

PROGETTO RETI WINSOME

Studente	Matricola
Luca Rizzo	598992

ANNO 2021/2022

Indice

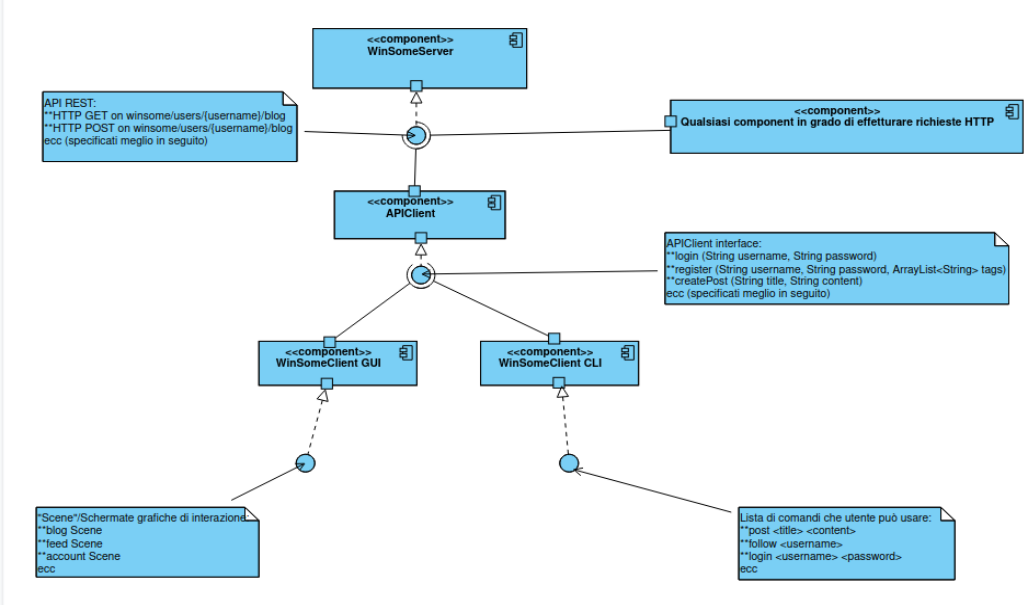
Introduzione	
1 SERVER	Pag. 5
1.1 RISORSE ESPOSTE E API REST	Pag. 5
1.2 RAPPRESENTAZIONE RISORSE	Pag. 9
1.3 GESTIONE CONCORRENZA SULLE RAPPRESENTAZIONI DELLE RISORSE	Pag. 9
1.4 ARCHITETTURA DEL SERVER	Pag. 10
1.5 RMI	Pag. 12
1.6 FILE DI CONFIGURAZIONE	Pag. 12
1.7 TERMINAZIONE SERVER	Pag. 13
2 APICLIENT	Pag. 14
3 INTERFACCE UTENTE	Pag. 17
3.1 CLI	Pag. 17
3.2 GUI	Pag. 18
4 PACKAGE RESTfulUtility e JsonUtility	Pag. 22
5 ISTRUZIONI PERCOMPILARE ED ESEGUIRE	Pag. 22
5.1 SERVER	Pag. 22
5.2 INTERFACCIA DA LINEA DI COMANDO	Pag. 23
5.3 INTERFACCIA GRAFICA	Pag. 23

Introduzione

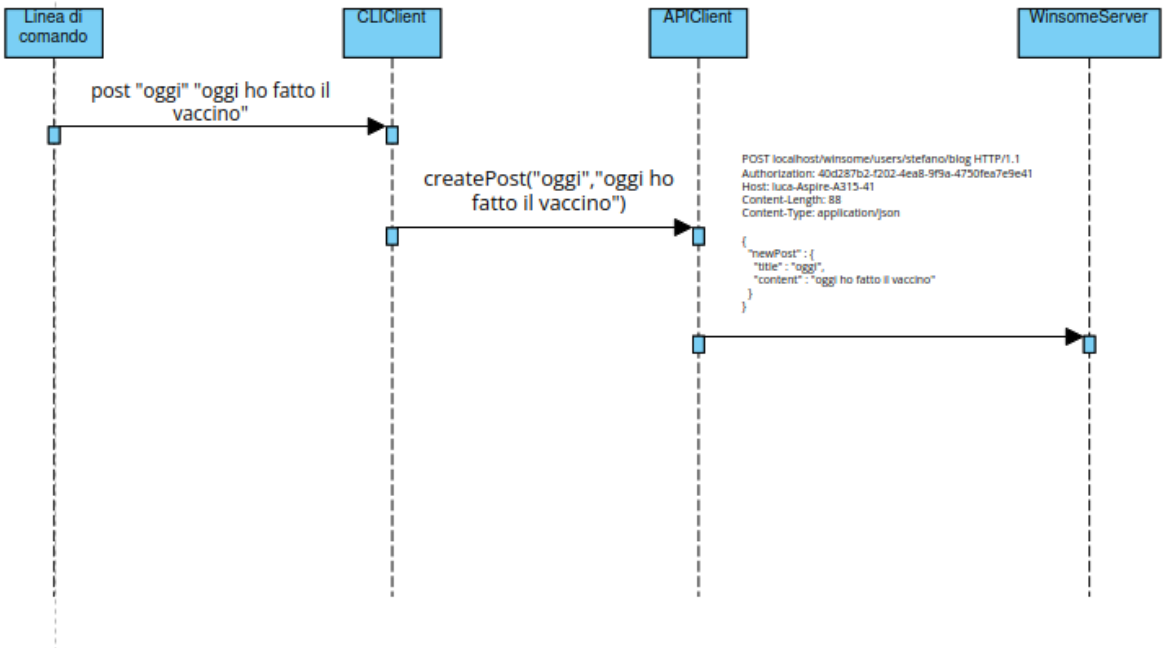
L'obiettivo del progetto era l'implementazione del social network WINSOME seguendo lo stile architetturale client-server ed il paradigma richiesta-risposta.
Lo sviluppo del social network è stato guidato dall'implementazione di 3 componenti interconnesse ciascuna con funzionalità ben definite:

1. *WinServer*: componente che gestisce tutte le risorse presenti nel sistema WINSOME. Si occupa di verificare che ciascuna richiesta sia autorizzata prima di modificare una risorsa o di mostrare informazioni relative ad una data risorsa
2. *APIClient*: componente intermedia che gestisce tutto il protocollo di comunicazione con il server WinServer. Tale componente mette a disposizione un'interfaccia chiara per modificare risorse nel server o per ottenere dati dal server senza che le varie interfacce di interazione con l'utente (GUI o CLI) si debbano preoccupare di gestire I dettagli tecnici della comunicazione.
3. *Interfacce di interazione con l'utente*: componente che permette all'utente di interagire con l'intero sistema tramite semplici comandi (CLI) o tramite un'interfaccia grafica

Le 3 componenti sono fortemente correlate, ma ciascuna di essa cerca di essere il più indipendente possibile dalle altre: ognuna effettua controlli sugli argomenti non curante del fatto che questi controlli possano essere stati effettuati già da un'altra componente. Ritengo che questa metodologia di progettazione abbia vantaggi a lungo termine, garantendo una maggiore mantenibilità del sistema e anche un maggior livello di interoperabilità. Come si vedrà nel corso della relazione, infatti, un utente potrebbe reperire informazioni dal server (o modificare informazioni presenti) anche attraverso un semplice browser visto che il server è stato sviluppato seguendo lo stile architetturale REST: indipendentemente dalla componente con cui comunica, il server effettuerà comunque tutti i suoi controlli di sicurezza.
Ciascuna componente comunica con l'altra attraverso un'interfaccia ben definita: *WinServer* comunica con *APIClient* esponendo delle risorse ed un API REST per operare su tali risorse con semantica ben definita; *APIClient* comunica con le interfacce di interazione con l'utente tramite un'interfaccia che espone metodi per ottenere in maniera semplice dati dal server, senza badare ai meccanismi di comunicazione. Segue un diagramma delle componenti che identifica per principali componenti del sistema e Ie loro interfacce di comunicazione.



L’utente medio che vuole usare il sistema *Winsome* interagirà tramite endpoint offerti dalla *WinsomeClient CLI* e *WinSomeClient GUI*. Segue un diagramma delle interazioni che mostra la serie di chiamate da ciascuna componente all’altra, scatenata in seguito al comando post “oggi” “oggi ho fatto il vaccino” passato da un utente da linea di comando dall’utente stefano che ha eseguito il login precedentemente:

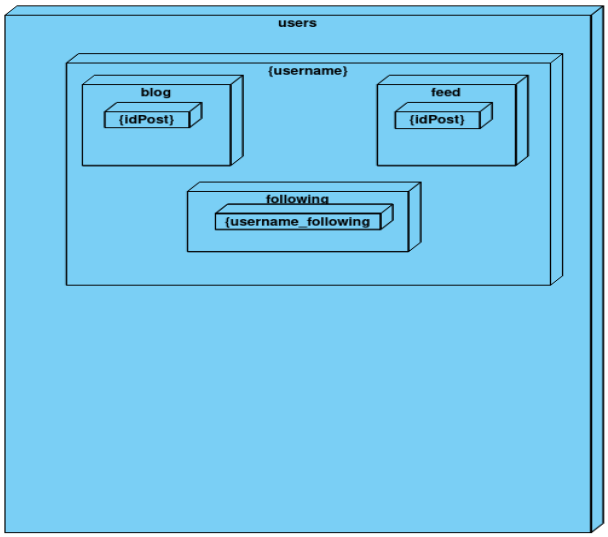


Segue la descrizione dettagliata di ciascuna componente.

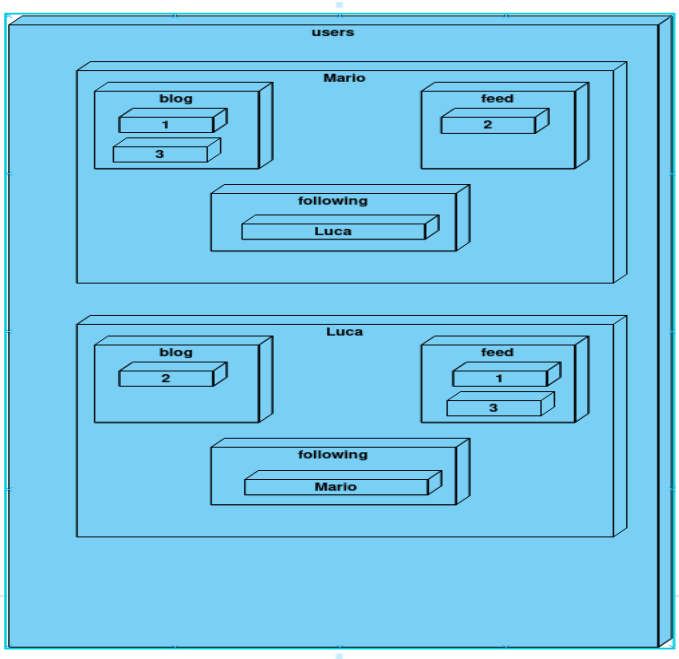
1 SERVER

1.1 *RISORSE ESPOSTE E API REST*

Il server è stato sviluppato secondo lo stile architetturale REST: in tal senso espone delle risorse a cui si può accedere tramite un identificativo ben definito (URI) e tramite i metodi HTTP. La semantica dei metodi sulle risorse è il più lineare possibile e segue il più possibile la semantica dei metodi HTTP. Di seguito una rappresentazione delle risorse che il server espone:



Segue un esempio di risorse esposte dal server nel momento in cui sono iscritti due utenti, Luca e Mario, che si seguono a vicenda e hanno pubblicato rispettivamente il post con id 2 e i post con id 1 e 3



Ciascun client, per interagire con il server, deve conoscere l'API REST esposta da WinServer. Di seguito è presente una semplice documentazione che indica i metodi consentiti e la semantica di ciascuna richiesta HTTP su una particolare risorsa esposta dal server.

URI	Risorsa identificata	GET	POST	PUT	DELETE
winsome/tokens	lista di token temporanei per utenti autorizzati (iscritti tramite RMI e che hanno effettuato login)	NO	SI; **Semantica**: crea un nuovo token/ effettua il login	NO	NO
winsome/tokens/{id-token}	specifico token relativo ad una sessione/login	NO	NO	NO	SI se si specifica api key relativa a token {id-token}; **Semantica**: eliminare un token ottenuto precedentemente/fare logout
winsome/users	lista utenti iscritti a WINSOME	SI se si specifica nell' header Authorization un api-key valido; **Semantica**: ottenere la lista utenti iscritti a Winsome. Si può inserire una query per ottenere utenti con particolari interessi	NO	NO	NO
winsome/users/{username-utente}	informazioni private su uno specifico utente	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: ottenere informazioni private di un utente	NO	NO	NO
winsome/users/{username-utente}/blog	blog relativo all' utente {username-utente}	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: ottieni solo la lista di post del blog di {username-utente}	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: pubblica un nuovo post nel blog di {username-utente}	NO	NO
winsome/users/{username-utente}/blog/{id-post}	post all'interno del blog dell' utente {username-utente}	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: ottieni contenuto e reazioni del post {id-post} nel blog di {username-utente};	NO	NO	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: eliminare il post {id-post} dal blog di {username-utente}
winsome/users/{username-utente}/feed	feed relativo all' utente {username-utente}	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: ottieni solo la lista di post del feed di {username-utente}	NO	NO	NO
winsome/users/{username-utente}/feed/{id-post}	post all'interno del feed dell' utente {username-utente}	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: ottieni contenuto e reazioni del post {id-post} nel feed di {username-utente};	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: aggiungere una reaction al post {id-post}	NO	NO
winsome/users/{username-utente}/following	lista di utenti che {username-utente} segue	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: ottenere lista utenti di cui {username-utente} è follower	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica**: seguire un nuovo utente	NO	NO
winsome/users/{username-utente}/following/{username-following}	identifica un particolare utente tra quelli seguiti da {username-utente}	NO	NO	NO	SI se si specifica nell' header Authorization l' api-key associato all'utente {username-utente}; **Semantica:** unfolloware utente {username-following}

La scelta dei metodi consentiti o meno su una data risorsa è guidata dalla struttura del progetto e rende semplice e intuitivo capire cosa è consentito fare nel mondo WINSOME: vedendo tale API-REST mi accorgo subito che non posso eseguire una POST su un post del mio blog (POST sulla risorsa winsome/users/{username}/blog/{id-post}), ovvero non posso aggiungere reaction ad un post del mio blog. Posso però farlo per un post nel mio feed (POST sulla risorsa winsome/users/{username}/feed/{id-post}). Posso eliminare un post nel mio blog (DELETE sulla risorsa winsome/users/{username}/blog/{id-post}) ma non un post del mio feed. Posso seguire un utente (POST sulla risorsa winsome/users/{username}/following) e unfolloware un utente (DELETE sulla risorsa winsome/users/{username}/following/{usernameUnfolloware}). L'accesso alla risorsa winsome/users tramite una GET può avvenire per mezzo di una query parametrica per specificare utenti solo con determinati interessi: ad esempio GET winsome/users?tag=calcio&tag=piscina

La fase di registrazione non è gestita tramite API REST perchè è stato richiesto esplicitamente nel testo del progetto l'utilizzo di RMI. In fase di registrazione, l'oggetto RMI esportato si occuperà di creare le risorse relative al nuovo utente registrato: gli assocerà un account identificato dall'URI *hostnameServer:porta/winsome/users/{username}*, un blog identificato dall'URI *hostnameServer:porta/winsome/users/{username}/blog*, un feed identificato dall'URI *hostnameServer:porta/winsome/*users*/{username}/feed* e un contenitore di utenti seguiti identificato dall'URI *hostnameServer:porta/winsome/users/{username}/following*.
Ovviamente queste rappresentano le risorse che l'utente percepisce vengano create e non la reale rappresentazione dello stato del server (descritto dopo).
Una volta che l'utente si è registrato, sa che tali risorse sono state create e vi interagisce tramite API REST. Per rispettare il principio STATELESS dello stile REST è stato introdotto il concetto di token: ogni volta che un utente vuole operare su una risorsa, deve effettuare il login attraverso una PUT sulla risorsa *winsome/tokens:* se l'utente era registrato, il server risponderà con un api-key che serve da chiave di autorizzazione per le richieste successive sulle risorse su cui si è identificato tramite username e password. Da quel momento in poi tutte le risorse relative all'utente sono associate all'api-key rilasciata: chi ne è in possesso è autorizzato ad accedere alle risorse e, se consentito, modificarle. Tutte le richieste del client includeranno tale api-key nell'header Authorization per indicare al server che egli è autorizzato ad accedere alle risorse associate a quell'api-key. In tal modo viene rispettato il principio STATELESS: ciascuna richiesta contiene al suo interno tutto il necessario per essere elaborata. La richiesta infatti contiene:

1. Un'operazione su una risorsa del server identificata tramite URI: es GET su *winsome/users/{username}/blog*.
2. Un'api-key associata ad un precedente login che autorizza l'accesso alle risorse relative a *{username}.* Il server, manterrà una relazione apiKey-utente, che indicherà quali sono le operazioni consentite ad un utente con una data apiKey. In generale, un apiKey è sempre associata a tutte le risorse di un particolare utente: il server permetterà la modifica delle risorse relative a *{username}* solo se si inserirà la corretta api-key rilasciata dal server a *{username}* in fase di login.

Ciascuna richiesta sarà perciò gestita secondo uno schema ben preciso:

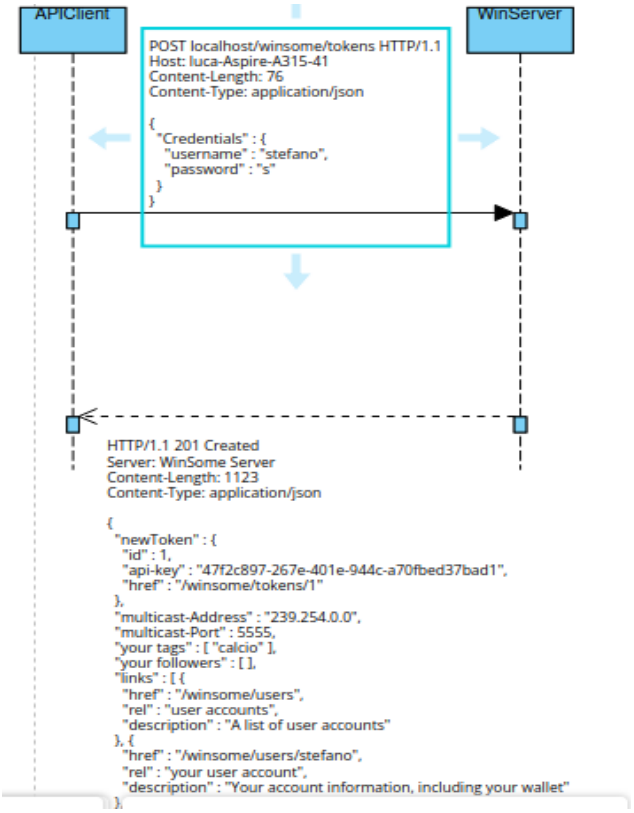
1. Dall'URL della richiesta individuo la risorsa a cui l'utente vuole accedere: tale risorsa (se l'utente è loggato) è associata ad una api-key. Solo inserendo la corretta api-key posso accedervi e, se consentito, modificarla
2. Dall'header "Authorization" prelevo l'api-key che ha inserito l'utente
3. Confronto l'api-key inserita dall'utente con quella associata alla risorsa: se sono le stesse, la richiesta è autorizzata e può essere effettuata

Questo meccanismo assume il ruolo di vera e propria autenticazione solo nel caso in cui la comunicazione TCP poggi su uno strato che ne garantisca riservatezza (es SSL/TLS): in questo modo username, password e apiKey private sono leggibili solo da client e server che comunicano e incomprensibili ad utenti malintenzionati. Non essendo argomenti trattati nel corso non sono stati introdotti nel progetto.
Il server riceverà la richiesta e potrà elaborarla in maniera indipendente dalle precedenti.
Un client, una volta che ha eseguito il login, riceve come risposta un messaggio con *statusCode = 201* e le seguenti informazioni in formato JSON:

- un apikey per eseguire le richieste,
- un riferimento ad un indirizzo di multicast e relativa porta per ottenere notifiche sul calcolo delle ricompense
- la lista di tag di suo interesse

- la lista dei suoi followers per inizializzare la sua struttura dati locale: se egli successivamente si registrerà al meccanismo di callback, otterrà anche le notifiche sui nuovi follower o su quelli che ha perso e aggiornerà la sua struttura dati locale (specificato meglio di seguito).

Il server, quando un utente esegue il login, invia anche una lista di riferimenti/URI a tutte le risorse a lui associate, per far si che non debba ricostruirli il client stesso. Otterrà, infatti, l'URI relativa al suo account, al suo blog, al suo feed e al suo following.
Segue un esempio di corretto login da parte dell'utente "stefano":



Non sono stati inclusi tutti i riferimenti mandati dal server per questioni di spazio

E' possibile, comunque osservare lo scambio richiesta-risposta dal terminale del server.
Tutti i corpi dei messaggi HTTP sono in JSON (specificato nell'header Content-type di ciascuna richiesta e risposta). Il server fa largo uso dei metodi statici offerti dalla classe *JsonMessageBuilder appartenente al package jsonUtility* per la creazione dei corpi delle risposte in formato JSON: questa classe fa uso della libreria Jackson per la creazione di tali messaggi in formato JSON. In particolare accederà alle rappresentazioni delle risorse per prelevare le informazioni da restituire ai vari client e le formatterà secondo la sintassi JSON. Questo meccanismo è thread-safe poiché l'accesso alle rappresentazioni delle risorse è stato reso thread-safe (spiegato meglio dopo).
Tutto ciò che ho appena descritto è lo schema di alto livello dal punto di vista dell'utente. Il server mantiene, infatti, rappresentazioni delle risorse che non hanno un rapporto 1:1 con le risorse esposte tramite l'API REST.

1.2 RAPPRESENTAZIONE RISORSE

La rappresentazione delle risorse esposte consta di due strutture dati concorrenti:

final private ConcurrentHashMap<String, User> registeredUser;
final private ConcurrentHashMap<Token, User> loggedUser;

La risorsa winsome/tokens è rappresentata da *loggedUser*: una PUT su tale risorsa crea una corrispondenza token e relativo utente. Una DELETE su un particolare token (su winsome/tokens/{id-token}) elimina un entry di tale Hash-Map. Il contenitore users è rappresentato dalla struttura dati *registeredUser.*
Tutte le risorse relative ad un utente sono rappresentate da un'istanza di User che contiene al suo interno:
final private ConcurrentHashMap<Long, Post> blog;
final private RWHashSet<User> followedUsers;
final private RWHashSet<User> followers;
final private Wallet wallet;
Il feed di un utente è costruito recuperando i blog di tutti gli utenti appartenenti a *followedUsers*. Il Set followers è presente anche lato server (nonostante richiesto dal progetto di gestirlo lato client tramite notifiche asincrone RMI-callback) affinchè un utente, una volta loggato, possa inizializzare la sua struttura dati locale che verrà poi aggiornata, come previsto dal progetto, tramite callback. Ciascun post è identificato da un idPost e mantiene, oltre alle informazioni base come titolo, contenuto e iterazioni per il calcolo delle ricompense, una struttura dati concorrente *RWHashSet<Reaction> reaction* per la memorizzazione delle reaction associate al post. Una reaction può essere di due tipi: like o commento. Un utente può associare ad un post del suo feed più commenti ma una sola reaction (il metodo equals() di reaction mostra quando due reaction collidono).
Gli attributi *long iterazione* di Post e *boolean isNew* di Reaction sono necessari per il calcolo delle ricompense.

1.3 GESTIONE CONCORRENZA SULLE RAPPRESENTAZIONI DELLE RISORSE

Tutte le risorse sono rappresentate attraverso strutture dati concorrenti e thread-safe: la modifica e l'accesso di *registeredUser* e *loggedUser* avvengono sempre attraverso i metodi atomici offerti dalle *ConcurrentHashMap*. Se le strutture dati esterne sono comunque thread-safe, particolare attenzione è stata rivolta nel rendere thread safe anche i valori associati alle varie chiavi delle *ConcurrentHashMap*, ovvero le istanze della classe User. Questa rappresenta il blog attraverso una *ConcurrentHashMap* (thread-safe) e mantiene due *RWHashSet*, ovvero un'implementazione concorrente della classe *HashSet*: permette a un solo scrittore di accedervi in mutua esclusione ma consente più lettori contemporaneamente. In particolare, quando è necessario iterare lungo tali collezioni (come fanno ad esempio I metodi statici di *JsonMessageBuilder*), si può richiedere *l'HashMap* che un *RWHashMap* gestisce: questa sarà uno "screenshot" dell'HashMap contenuta nella *RWHashMap*, in modo che ciascun thread che vuole iterare lungo tali strutture non debba badare a meccanismi di concorrenza. Meccanismo simile viene usato nel ritornare il feed di un utente: si itera lungo ciascun utente dei *followedUsers* (sempre la copia ritornata da *RWHashSet*) e si aggiunge ciascuna entry del blog in una struttura dati che sarà ritornata solamente al thread richiedente e quindi non condivisa. Chi accede e modifica tali oggetti non deve perciò preoccuparsi di gestire la concorrenza ma li utilizza in maniera trasparente come se si trattasse di oggetti non condivisi. Il wallet associato a ciascun utente è anche thread safe tramite l'uso di una *RWLock*: non ci saranno problemi se un utente richiede il suo wallet mentre un thread sta eseguendo il task *RewardCalculator* che modifica il wallet.

Ciascun post, a sua volta, mantiene le reaction in un *RWHashSet*. Tutte le istanze di classi che contengono attributi non *final* mantengono anche una *ReentrantReadWriteLock stateRWlock* per l'accesso e la modifica concorrente dei campi (ad esempio iterazioni di Post e isNew di reaction che possono essere accedute dal thread che calcola le ricompense in scrittura, e dal thread per effettuare il backup periodico in lettura).
Ho optato per l'utilizzo di una *ConcurrentHashMap* rispetto ad una *HashMap*, che sincronizza tutta la struttura dati in scrittura, per rendere maggiore il livello di concorrenza a discapito di una perfetta sincronizzazione delle risorse: se, ad esempio, un utente A segue un utente B, e mentre un thread worker sta accedendo al blog di B (tramite iteratore sulla *ConcurrentHashMap* che rappresenta il blog di B) poichè A ha richiesto il suo feed, B pubblica un nuovo post, non siamo sicuri che tale post sarà incluso nel feed di A. Questo a causa del fatto che l'iteratore può o meno catturare le modifiche (weakly consistent iterator): tale scenario non è un errore perchè l'utente può in qualsiasi momento ricaricare il suo feed e ottenere il nuovo post pubblicato da B. Una sincronizzazione delle risorse troppo elevata sarebbe perciò controproducente e possiamo quindi accettare piccoli compromessi sulla semantica delle operazioni in tale contesto, per migliorare l'accesso concorrente alle risorse.

1.4 ARCHITETTURA DEL SERVER

Per lo sviluppo del server ho adottato l'approccio NIO + threadpool di worker per la gestione delle richieste e la preparazione delle risposte. La scelta è stata dettata dalla natura del progetto: un utente, nell'utilizzo di un social network è nella stragrande maggioranza dei casi: o inattivo perchè sta, ad esempio, leggendo il contenuto di un post, oppure scambia comunque ad intervalli irregolari piccole quantità di dati. Un approccio un client per connessione rappresenterebbe un enorme spreco di risorse in continuo ascolto in connessioni con un traffico dati irregolare e scarso. Nel mio approccio si possono gestire un numero relativamente infinito di client contemporaneamente (molti dei quali anche inattivi) non dettato dalla memoria del computer: ciò non è possibile con un approccio un thread per connessione (infatti esiste un numero massimo di thread attivabili, dettato dalla memoria del computer poichè ciascun thread alloca delle risorse in memoria principale). Si potrebbe optare per l'utilizzo di un cachedThreadPool con il modello un thread per connessione ma ciò limiterebbe il numero massimo di utenti connessi contemporaneamente al numero di thread del threadpool.
Nel mio approccio ho un thread dispatcher che esegue tutte le operazioni di network I/O (lettura e scrittura attraverso i canali in modalità non bloccante) e un threadpool per la gestione delle richieste e la produzione di risposte. Il thread dispatcher legge le richieste dei client in un *ByteBuffer* attraverso i channel associati alle *SelectionKey*, passa la richiesta ad un worker e attende che quest'ultimo la processi e produca una risposta. Nel frattempo può continuare a svolgere altre mansioni: leggere nuove richieste; inviare risposte a precedenti richieste; registrare interesse chiavi in scrittura. Quando il worker ha preparato la risposta, avverte il thread dispatcher che una risposta, "attaccata" alla *SelectionKey*, è pronta per essere inviata attraverso il channel associato a tale *SelectionKey*. La comunicazione con i client è tutta ad opera del thread dispatcher: ciò non è un problema poichè la quantità di dati che il server e il client si scambiano è poca e comunque un solo thread alla volta potrebbe accedere alla scheda di rete (inoltre il processore è molto più veloce della scheda di rete e leggere attraverso un channel in modalità non bloccante è computazionalmente molto veloce). Il thread dispatcher è quindi specializzato nel network I/O. Esso si attiverà solo quando vi è qualcosa da leggere o scrivere nei channel. Riusciamo, in questo modo, a risparmiare sul numero di thread attivi, riducendo sia l'overhead dovuto al content-switch se vi sono moltissimi thread attivi, sia l'appensantimento della memoria causato da ciascun thread attivo.
In generale il thread dispatcher resta in attesa di leggere una richiesta HTTP completa da un client e, una volta letta, la passa ad un worker del threadPool (rappresentata dalla stringa *fromClient* prelevata dal ByteBuffer). Un thread del threadpool eseguirà il task *RequestHandler*, ovvero gestirà la richiesta e creerà

la risposta risposta relativa a tale richiesta. Notificherà, infine il thread dispatcher che la richiesta è pronta e che può inviarla attraverso il channel corrispondente.
Lo schema per la gestione della richiesta è il seguente:

1. Thread dispatcher accetta una nuova connessione, la aggiunge al selector registrando come operazione di interesse la lettura e associando alla *SelectionKey* un *ByteBuffer* per la lettura
2. Thread dispatcher legge un'intera richiesta HTTP attraverso il chennel associato alla *SelectionKey* nel *ByteBuffer* "attaccato", se ne disinteressa (setta le operazioni di interesse per *la* data *SelectionKey* a 0, ovvero nessuna operazione di interesse) e crea un nuovo *RequestHandler* che sarà associato a tale *SelectionKey* e alla particolare richiesta presente nella stringa *fromClient*
3. Un thread del threadpool eseguirà il task *RequestHandler* e gestirà la richiesta, preparando una risposta
4. Una volta che il thread worker ha preparato la risposta, la inserirà in un *ByteBuffer* e associerà alla *SelectionKey* tale *ByteBuffer*. Successivamente aggiungerà la *SelectionKey* ad una lista che individua tutte le *SelectionKey* che sono pronte in scrittura (*ArrayList<SelectionKey> readyToWrite*) e eseguirà una wakeup sul *Selector* del thread dispatcher per notificarlo di nuove chiavi pronte in scrittura. Tale meccanismo è necessario poichè solo in questo modo si eliminano tutte le race condition relative al cambio di operazioni di interesse di una selectionKey durante la *select()* del relativo *Selector* (in alcune implementazioni il metodo *interestOps()* si blocca viene eseguito durante la *select()* del relativo *Selector*). Maggiori dettagli:

   SelectionKey JavaDoc says this:

   > The operations of reading and writing the interest set will, in general, be synchronized with certain operations of the selector. Exactly how this synchronization is performed is implementation-dependent: In a naive implementation, reading or writing the interest set may block indefinitely if a selection operation is already in progress; in a high-performance implementation, reading or writing the interest set may block briefly, if at all.

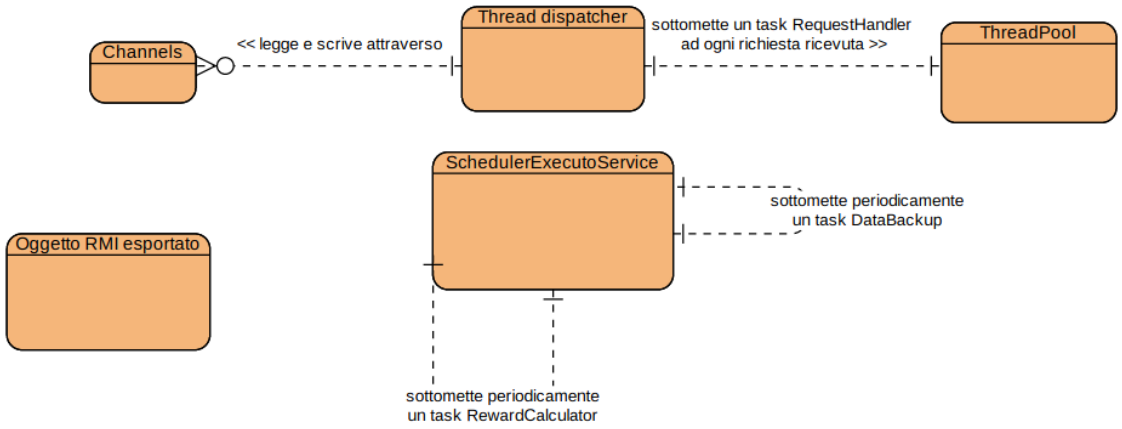
   In questo modo solo il thread dispatcher cambierà operazioni di interesse relative ad una *SelectionKey* e non si avranno race condition: infatti, prima si cambiano operazioni e poi si fa la select o viceversa (mai contemporaneamente).
5. Il thread dispacher viene svegliato dalla wake-up, scorre la lista delle *SelectionKey* pronte in scrittura e ne registra l'interesse per la scrittura
6. Scrive attraverso il channel associato alla *SelectionKey* il contenuto del ByteBuffer "attaccato"

I thread worker stamperanno tutte le richieste ricevuto e le risposte generate sul terminale così che sia possibile vedere lo scambio richiesta risposta tra client e server.
Oltre al thread dispatcher e al thread pool, il server attiverà, a intervalli regolari tramite l'utilizzo di uno *ScheduledExecutorService,* un thread per la gestione delle ricompense ed un thread per il backup periodico delle risorse su disco tramite i task *RewardCalculator* e *DataBackup*.
Il thread per la gestione delle risorse scorrerà semplicemente tutti i post presenti nei vari blog, ne calcolerà per ciascuno le ricompense, le dividerà tra curatori e autore e aggiornerà il wallet (oggetto thread-safe) di ciascun utente.
Il thread per il backup periodico scorrerà tutte le rappresentazioni delle risorse e le salverà su disco facendo uso della libreria Jackson per formattare i dati nel formato JSON prima di salvarli in un file di backup. Non effettua alcun meccanismo di caching: salva sempre tutte le risorse presenti in memoria principale su disco.
All'avvio successivo il server controllerà se è presente il file di backup, il cui path è specificato nel file di configurazione, e ripristinerà lo stato del server scorrendo il file e caricando in memoria principale tutte

le informazioni salvate (meccanismo inverso compiuto dal task *DataBackup*). In tal modo il processo server può essere interrotto e ripreso senza perdita di dati e in maniera trasparente al client.

1.5 RMI

L'oggetto RMI creato, esportato e registrato nel registry col nome "RMI-WINSERV" (si può modificare dal file di configurazione del server e del client) consentirà ai client di registrarsi a WINSOME e, una volta loggati, di registrarsi ad un servizio di callback per notifiche asincrone su nuovi follower. Il meccanismo RMI attiverà uno o più thread per la gestione delle chiamate sull'oggetto remoto: i metodi che implementano l'interfaccia *IntRMIServer* accedono a strutture dati thread-safe e sono, quindi, globalmente thread-safe. Infatti, quando un utente si registra, viene aggiunto alla struttura dati condivisa *registeredUsers* e, se un utente si registra al meccanismo di callBack verrà aggiunto, insieme all'oggetto remoto di callback passato dal client, alla struttura dati *usersToNotify* e notificato tramite le chiamate all'oggetto remoto di callback *notifyNewFollower(String usernameNewFollower)* e *notifyLostFollower(String usernameLostFollower)* se un utente smette o inizia a seguirlo.
Per rendere il protocollo di comunicazione più coerente possibile, i metodi ritorneranno stringhe che contengono risposte HTTP dalle quali possiamo ottenere tutte le informazioni sui risultati delle operazioni.
Segue uno schema generale dell'architettura di WinsomeServer:



1.6 FILE DI CONFIGURAZIONE

Il file di configurazione serve per avviare il server e contiene tutte le informazioni per il corretto funzionamento del server. Esso deve essere in formato JSON con coppie "parametri" : valori per ciascun parametro necessario all'avvio del server. In particolare esso specifica:

- Indirizzo multicast e porta a cui il thread avviato per il calcolo delle ricompense invia le notifiche di aggiornamento. Tale indirizzo verrà inviato al client in seguito alla login, cosicché il client saprà a quale gruppo multicast iscriversi (spiegato meglio dopo)
- Indirizzo IP e porta della listenSocket per avviare connessioni TCP con I vari client che vogliono loggarsi
- Porta associata al registry e nome dell'oggetto remoto esportato per la registrazione degli utenti e la registrazione al meccanismo di callback
- timerRewardMin: intervallo di tempo tra un calcolo delle ricompense e l'altro

- authorPercentage: percentuale di ricompensa da assegnare ad un autore sul guadagno totale calcolato per ciascun post. La ricompensa per ciascun singolo curatore sarà pari a (100-authorPercentage)/n_curatori
- timerBackupMin: intervallo di tempo tra un backup su disco e l'altro nel file individuato da B
- backupJsonFile
- backupJsonFile: path del file JSON di backup
- nworker: numero di worker per la gestione delle richieste

Segue un esempio di file di configurazione. Esso è presente nella cartella ./resource: si consiglia di modificare solamente i valori associati alle stringhe se necessario.

```
{
  "multicastAddress" : "239.254.0.0",
  "multicastPort" : 5555,
  "registryPort" : 6788,
  "rmiobjectbindingName" : "RMI-WINSERV",
  "ip" : "localhost",
  "tcpport" : 6789,
  "timerRewardMin" : 5,
  "authorPercentage" : 70,
  "timerBackupMin" : 7,
  "backupJsonFile" : "./resource/usersResourceBackup.json",
  "nWorker" : 5
}
```

All'avvio, WinsomeServer necessita del path di questo file di configurazione per ricreare un'instanza corretta della classe *ServerConfiguration*. Un formato errato di tale file genererà un errore e non consentirà l'avvio del server.

1.7 TERMINAZIONE SERVER

WinsomeServer termina quando viene inserito da linea di comando la stringa "exit". In tal caso, il main thread eseguirà *l'interrupt()* e il *join*() di winsomServer. Una volta interrotto, winsomeServer eseguirà la terminazione "gentile" dei vari threadpool (quello per i worker e quello per i task da schedulare ad intervalli regolari), esegue unbind dal registry e "unexoport"dell'oggetto RMI esportato e ed esegue il task di backup finale per salvare ultimi aggiornamenti su disco. In tal modo tutti i threads attivi termineranno e il server verrà chiuso in maniera "pulita".

2 APICLIENT

Questa componente è responsabile di realizzare la comunicazione con il server e esporre alle interfacce di interazione utente metodi trasparenti che ritornano semplici risposte HTTP. Egli si occuperà di tutti i dettagli tecnici. Espone la seguente interfaccia Java:

```java
public interface APIClientInterface {
    HttpResponse register(String username, String password, ArrayList<String> tags) throws RemoteException, NotBoundException, ParseException, Unsuppor
    HttpResponse login(String username, String password) throws IOException, ParseException;
    HttpResponse accountPrivateInfo() throws IOException, ParseException;
    HttpResponse viewBlog() throws IOException, ParseException;
    HttpResponse createPost (String title, String content) throws IOException, ParseException;
    HttpResponse rewinPost (String idPost) throws IOException, ParseException;
    HttpResponse deletePost ( String idPost ) throws IOException, ParseException;
    HttpResponse logout() throws IOException, ParseException, InterruptedException;
    HttpResponse followUser(String username) throws IOException, ParseException;
    HttpResponse unfollowUser(String username) throws IOException, ParseException;
    HttpResponse viewFeed() throws IOException, ParseException;
    HttpResponse ratePost(String idPost, String rate) throws IOException, ParseException;
    HttpResponse addComment(String idPost, String content) throws IOException, ParseException;
    HttpResponse showPost(String idPost) throws IOException, ParseException ;
    HttpResponse listUsers() throws IOException, ParseException;
    HttpResponse listFollowing() throws IOException, ParseException;
    HashSet<String> listFollowers();
}
```

APIClient ha bisogno di un file di configurazione per essere inizializzata: esso deve essere in formato JSON e contenere l'insieme minimo delle informazioni necessarie all'APIClient per iniziare la comunicazione con il server. In particolare necessita di:

- Indirizzo IP e porta del server: necessari per instaurare la connessione TCP col server in seguito ad una richiesta di login. Tutte le successive richieste al server avverranno tramite tale connessione.
- Porta del registry del server e nome dell'oggetto RMI esportato dal server per la registrazione a WINSOME e al servizio di callback
- Nome dispositivo interfaccia di rete: necessario per iscriversi al gruppo Multicast e ricevere messaggio UDP di notifica sul calcolo delle ricompense

Segue un esempio di file di configurazione. Esso è presente nella cartella ./resource: si consiglia di modificare solamente i valori associati alle stringhe se necessario.

```json
{
  "serverIP" : "localhost",
  "listenSocketPort" : 6789,
  "serverRegistryPort" : 6788,
  "rmiobjectbindingName" : "RMI-WINSERV",
  "nameOfNetworkInterfaceDevice" : "wlp2s0"
}
```

Si tratta dell'insieme minimale di informazioni necessare per svolgere il suo compito: il resto delle informazioni verranno fornite dal server al momento del login. In seguito alla chiamata *login()*, il client apre una connessione TCP con la listenSocket del server "serverIP" in ascolto sulla porta "listenSocketPort" ed esegue una richiesta di PUT sulla risorsa winsome/tokens(unico URI delle risorse esposte dal server che deve conoscere) per ottenere tutte le informazioni necessarie per procedere nella comunicazione. Infatti egli riceverà in seguito al login:

- Indirizzo di multicast e porta per ottenere notifiche calcolo ricompensa
- Lista followers per inizializzare struttura dati locale che verrà, dopo che l'utente si iscrive al servizio di callback, aggiornata col meccanismo delle callback

- Api-key associata alle risorse relative all'utente: egli conserverà tale apiKey e la allegherà in ogni richiesta successiva nell'header "Authorization"
- Lista di riferimenti alle risorse appartenenti all'utente per le successive richieste al server.

Dopo aver ottenuto tutte le informazioni necessarie dal server in seguito alla login, il client avvia il thread *receiveNotificationThread* e ne mantiene un rifermento per poterlo successivamente interrompere ed effettuare il join. Il thread *receiveNotificationThread* resta costantemente in ascolto sulla MulticastSocket che si costruisce sulla base degli argomenti inviati dal server e stampa, ogni volta, il messaggio ricevuto.
Successivamente APIClient creerà un oggetto per il callback, lo esporterà ottenendo uno stub e si registrerà al servizio di callback grazie al metodo *registerForCallbackFollower()* dell'oggetto RMI esportato dal server, al quale passerà lo stub appena ottenuto. Le chiamate al metodo *notifyNewFollower() e notifyLostFollower()* saranno gestite dal meccanismo RMI ma sono thread-safe in quanto semplicemente stampano sul terminale una stringa e aggiornano la struttura dati locale followers che è thread-safe per i motivi già sopra elencati.
In questo modo, l'utente riesce, con il metodo *listFollowers()* ad ottenere la lista dei suoi followers, senza effettuare alcuna chiamata al server ma esaminando solamente la struttura dati locale.
Tutti i metodi non fanno altro che creare richieste HTTP (tramite la classe *HttpRequest* del package *RESTfulUtility*), scriverle attraverso la clientSocket di comunicazione col server e attendere una risposta HTTP che viene ottenuta tramite il metodo statico *HttpResponse.newHttpResponseFromStream(clientSocket.getInputStream())* della classe *HttpResponse* del pacakge *RESTfulUtility*.
APIClient formatta il corpo di qualsiasi richiesta con la sintassi JSON. Seguono una serie di esempi di messaggi di richiesta dell'APIClient in seguito a chiamate dei metodi (supponendo che l'utente "franco" si sia loggato e abbia ottenuto un'api-key):

- *ApiClient.createBlog("bella giornata", "oggi è una giornata magnifica! sono proprio felice")* genera tale richiesta HTTP

```
POST localhost/winsome/users/franco/blog HTTP/1.1
Authorization: bbc6f5d8-4de4-4c7d-babd-578d597126022
Content-type: application/json
Host: luca-Aspire-A315-41
Content-Length: 124

{
  "newPost" : {
    "title" : "bella giornata",
    "content" : "oggi è una giornata magnifica! sono proprio felice"
  }
}
```

- *ApiClient.followUser("b")* genera tale richiesta HTTP:

```
POST localhost/winsome/users/franco/following HTTP/1.1
Authorization: bbc6f5d8-4de4-4c7d-babd-578d597126022
Content-type: application/json
Host: luca-Aspire-A315-41
Content-Length: 49

{
  "userToFollow" : {
    "username" : "b"
  }
}
```

- *ApiClient.createPost("48","ciao")* genera tale richiesta HTTP:

```
POST localhost/winsome/users/franco/feed/48 HTTP/1.1
Authorization: bbc6f5d8-4de4-4c7d-babd-578d597126022
Accept: application/json
Host: luca-Aspire-A315-41
Content-Length: 63

{
  "reaction" : {
    "type" : 1,
    "content" : "ciao"
  }
}
```

- *ApiClient.deletePost("67")* genera tale richiesta HTTP:

```
DELETE localhost/winsome/users/franco/blog/67 HTTP/1.1
Authorization: bbc6f5d8-4de4-4c7d-babd-578d597126022
Host: luca-Aspire-A315-41
```

- *ApiClient.listUsers()* genera tale richiesta HTTP:

```
GET localhost/winsome/users?tag=calcio&tag=piscina HTTP/1.1
Authorization: bbc6f5d8-4de4-4c7d-babd-578d597126022
Accept: application/json
Host: luca-Aspire-A315-41
```

Nell'ultima richiesta includiamo nell'URL anche una query per specificare gli utenti con quali interessi vogliamo ottenere (quelli uguali ai nostri)
Da notare che tutte le richieste contengono l'api-key nel campo Authorization **per identificare il client nel server e autorizzare le operazioni.**
L'approccio IO bloccante è completamente in linea con il protocollo richiesta-risposta: una volta inviata una richiesta rimango in attesa di una risposta perchè il server ne invierà una (NOTA: lo stesso ragionamento non può essere adottato dal server che non sa quando riceverà una richiesta dal client e occuperebbe un thread in attesa passiva. In quel caso, visto l'intervallo di tempo irregolare tra le varie richieste inviate da un client, l'approccio NIO non bloccante risulta una strategia più azzeccata).
Ottenuta la risposta la passerà all'interfaccia utente che avrà il compito di renderla leggibile all'utente medio che non conosce JSON e il protocollo HTTP.

In seguito ad un logout viene chiamato il metodo *resetAll():* esso chiude la connessione TCP attiva con il server, esegue "unexport" dell'oggetto esportato per il callback, interrompe ed esegue il join del thread in ascolto per la recezione delle notifiche e resetta tutti i parametri non final dell'oggetto. Per interrompere il thread in ascolto su una MulticastSocket è stato necessario riscrivere il metodo *interrupt()* del thread: esso chiuderà la socket in modo da sollevare un eccezione nel thread in attesa di ricevere il messaggio, far stampare al thread un messaggio di terminazione sulla console e far terminare il thread in maniera "pulita". In questo modo l'istanza di APIClient non gestisce più altri thread e qualsiasi altra componente che fa uso di tale istanza può terminare senza curarsi di meccanismi per la gestione dei thread attivati da APIClient.

3 Interfacce utente

Esse rappresentano l'endpoint di interazione con l'utente medio che vuole utilizzare il sistema WINSOME. Sia la CLI che la GUI sfruttano l'interfaccia esposta da APIClient per ottenere informazioni dal server e gestirle secondo i loro scopi.

3.1 CLI

Essa rappresenta una semplice interfaccia da linea da comando attraverso la quale l'utente può interagire con il sistema WINSOME. L'utente inserisce i comandi da linea di comando e ottiene risposte attraverso la linea di comando. L'utente può interagire attraverso dei semplici comandi:

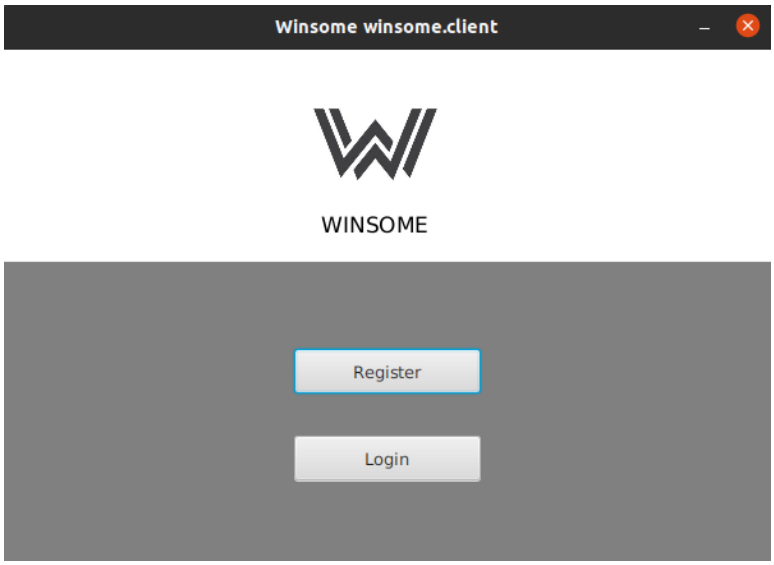
- register <username> <password> <tag separati da spazio>: per registrarsi al sistema WINSOME
- login <username> <password>: per effettuare login al proprio account
- logout: per eseguire il logout dall'account a cui si era precedentemente loggato
- blog: per ottenere la lista di post del blog. Per ciascun post viene mostrato id del post, titolo e autore
- post "<title>" "<content>": per pubblicare un nuovo post sul tuo blog
- show feed: per ottenere la lista di post del feed
- show post <idPost>: per ottenere titolo,contenuto,autore,reazioni del post <idPost>
- delete <idPost>: per eliminare <idPost> dal tuo blog
- rewin <idPost>: per pubblicare il <idPost> post presente nel tuo feed nel tuo blog
- comment <idPost> "<content>": per commentare con <content> il post <idPost>
- follow <usernameUtente>: per followare <usernameUtente>
- unfollow <usernameUtente>: per unfolloware <usernameUtente>
- rate <idPost> <vote>: per votare il post <idPost>. Il valore di <vote> può essere o 1 o –1
- list followers: per ottenere la lista dei propri followers
- list following: per ottenere lista degli utenti che segui
- list users: per ottenere lista degli utenti con i tuoi stessi tags
- wallet: per ottenere il valore del proprio wallet in winsome e la storia delle transizioni del portafoglio
- wallet btc: per ottenere il valore del portafoglio in winsome e in BTC
- exit per terminare il programma

Se l'utente inserisce un comando non presente tra quelli elencati sopra ottiene la lista dei soli comandi consentiti.

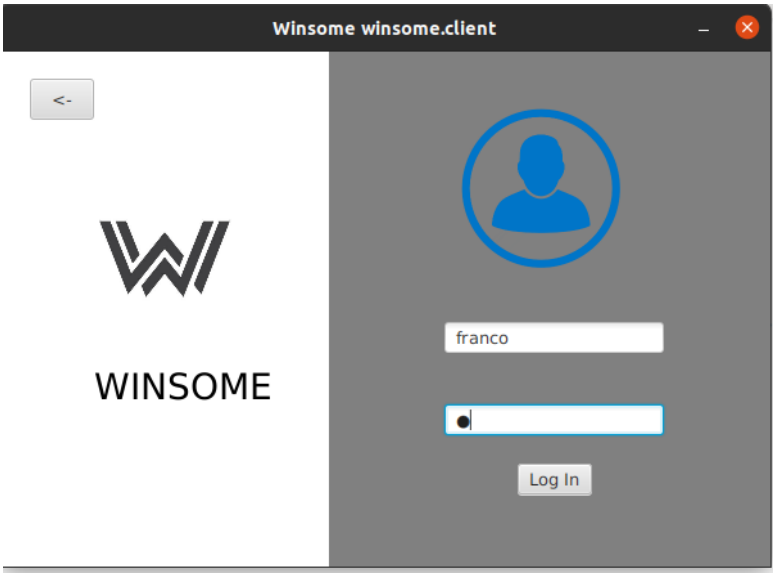
Prima di eseguire una chiamata a APIClient la componente CLIClient esegue controlli sugli argomenti passati da linea di comando per evitare richieste inutili al server: ad esempio verifica che il valore idPost sia un long; verfica che rate sia uguale a 1 o –1 ecc
Se gli argomenti sono errati, CLIClient stampa subito un messaggio di errore senza eseguire alcuna chiamata ad APIClient. Stessa cosa se l'utente prova ad eseguire un qualsiasi comando consentito prima di effettuare il login.
In seguito al comando exit, se l'oggetto apiClient che è stato istanziato è attivo in una sessione (avevamo eseguito login) viene effettuato il logout in modo da terminare in maniera "pulita" tutti i thread attivati da apiClient (sopra è spiegato il comportamento del logout).

3.2 GUI

Essa rappresenta una semplice interfaccia grafica per l'uso più intuitivo del sistema WINSOME. E' stata realizzato utilizzando la libreria javaFX presente nella directory ./lib a partire dalla directory base del progetto. Si compone di un unico Stage e varie Scene che variano a seconda dei bottoni cliccati e permette una completa interazione con il server. Le uniche funzionalità non supportate graficamente sono le notifiche delle ricompense e le notifiche relative a nuovi followers: le notifiche non compariranno nell'interfaccia grafica ma solo nella linea di comando come semplici stringhe (in realtà è la componente APIClient a stamparle attraverso i thread che attiva e l'oggetto remoto che esporta).
Ritengo che tale interfaccia di interazione renda più fruibile il concetto di risorse associate all'utente e operazioni consentite sulle risorse disponibili: egli infatti può navigare attraverso il suo blog o il suo feed in maniera intuitiva e sono evidenti e univoche le operazioni effettuabili. Come si vedrà meglio in seguito, la schermata di visualizzazione di un post del blog è differente rispetto a quella di visualizzazione di un post del feed, in quanto sono differenti le azioni consentite su uno e l'altro; posso unfolloware solo utenti che sono presenti nella mia lista following perchè solo loro avranno un bottone unfollow associato ecc.
Ciò porterà l'utente a sbagliare meno ed egli si sentirà guidato nell'uso del prodotto.
Ogni Scene è creata attraverso un file .fxml e gestita da un particolare Controller che regola il comportamento di ciascuna componente/controllo all'interno della Scene. Il Controller associerà a ciascuna componente (quasi tutte) una chiamata attraverso un' istanza di APIClient che restituirà informazioni per "riempire" l'interfaccia grafica. Nel passaggio da una Scene ad un altra viene caricato un nuovo file .fxml che inizializzerà una nuova Scene, la quale verrà settata nello Stage relativo all'applicazione. I vari Controller si passano sia un riferimento allo Stage corrente, sia un rifermiento all'oggetto APIClient istanziato inizialmente.
All'avvio, verrà caricata la Scene startScene.fxml attraverso *FXMLLoader* e il controller associato sarà lo *StartController*. Ad esso viene passata una istanza di APIClient che sarà la stessa lungo tutta l'esecuzione.



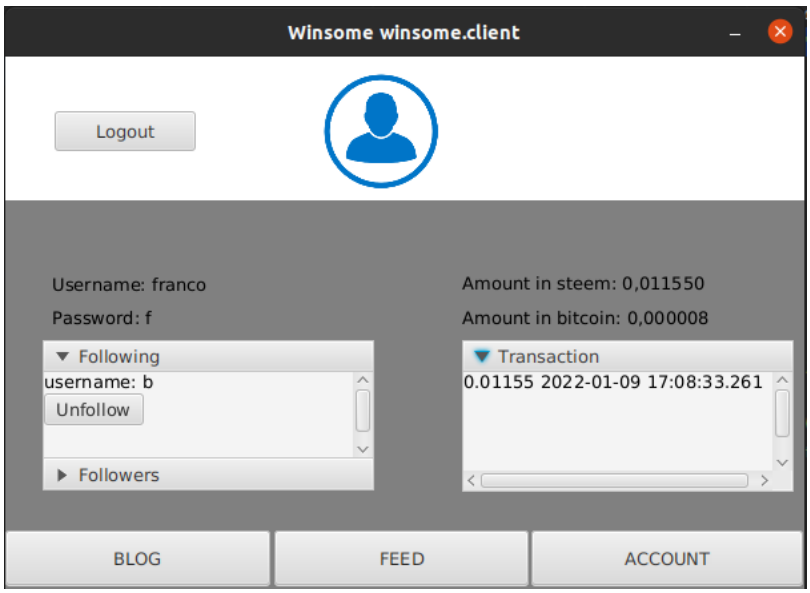
Successivamente, cliccando uno dei due bottoni si passerà alla fase di registrazione e login. Descriviamo, per brevità solamente il login.
Una volta cliccato il pulsante di login ci sposteremo in una nuova Scene con un nuovo controller *LoginController*.



Il LoginController associerà al bottone "Log in" la chiamata al metodo *login(username,password)* dell'APIClient, prelevando gli argomenti username e password dai text field associati. In caso di corretto login ci sposteremo nella BlogScene gestita dal *BlogController*. Egli inizializzerà la lista di post attraverso la chiamata *viewBlog()* all'APIClient e aggiungerà le informazioni sui post ottenute alla componente VBOX che rappresenta la lista di post del blog.

Il bottone "Create new post" sarà associato alla chiamata del metodo *createPost(title, content)* con gli argomenti title e content ottenuti dal relativo textField e textArea. Questo bottone aggiornerà anche la lista di post del blog attraverso una chiamata *viewBlog()* all'APIClient.
Il meccanismo è pressoché simile per le Scene Feed e Account: vengono inizializzate e aggiornate tramite i dati ottenuti attraverso chiamate ad un'istanza di APIClient che viene passata da controller a controller.

- AccountScene: è possibile controllare il proprio wallet, effettuare logout e visualizzare lista followers e following e unfolloware un utente che segui. All'avvio tutte le *Accordition* sono riempite tramite chiamate all'oggetto apiClient passato come argomento al costruttore di *AccountController*. Si può notare come un utente non possa unfolloware qualsiasi utente ma solo quelli nella lista following: in tal modo evitiamo il comportamento errato di unfollow di un utente che non seguivamo.



- FeedScene: è presente la lista dei post del feed e la lista di utenti con I tuoi stessi interessi. Tali utenti possono essere seguiti tramite il bottone “Follow”. Anche in questo caso ciascun bottone effettua chiamate all’istanza apiClient passata al costruttore di *FeedController*

Per concludere volevo evidenziare la differenza tra la scene BlogPost e la scene FeedPost: nella prima l’utente può solo visualizzare like e commenti ed, eventualmente eliminare il post col comando “delete post” (da notare come queste siano tutte le azioni consentite su un post del blog); nella seconda posso aggiungere un voto, effettuare il rewin o commentare il post. ***L’utente, dalla semplice schermata, comprende cosa è consentito e cosa non è consentito fare (definito dalla specifica del progetto) sulle sue risorse!***

- FeedPost Scene: post pubblicato da stefano e visualizzato dall’account di franco

- BlogPost Scene: post pubblicato da stefano visualizzato dall'account di stefano

4 PACKAGE RESTfulUtility e JsonUtility

Si tratta di due semplici package per la semplificazione di alcuni compiti: il primo contiene le classi HttpResponse e HttpRequest (da me implementate) per la gestione dei messaggi che client e server si scambiano e le classi Token e Link le cui istanze rappresentano rispettivamente un token usato per l'autorizzazione di una richiesta per una risorsa e link ad una risorsa esposta dal server.
Il secondo package contiene solo la classe JsonMessageBuilder che possiede metodi statici per la formattazione dei dati presenti nel server e da inviare al client in formato JSON. I metodi fanno uso della libreria Jackson fornita a lezione.

5 ISTRUZIONI PER COMPILARE ED ESEGUIRE

5.1 Server

Per compilare il server è necessario eseguire il seguente comando (ambiente Linux) a partire dalla root directory del progetto:

- *javac -cp lib/jackson-annotations-2.9.7.jar:lib/jackson-core-2.9.7.jar:lib/jackson-databind-2.9.7.jar src/winsome/WinServerMain.java src/winsome/client/*.java src/winsome/jsonUtility/*.java src/winsome/resourseRappresentation/*.java src/winsome/RESTfulUtility/*.java src/winsome/server/*.java -d out/production*

In questo modo compiliamo tutti i file .java necessari e includiamo le librerie Jackson utilizzate per formattare i dati in formato JSON e indirizziamo tutto l'output prodotto nella cartella out/production.
Per eseguire il server è necessario eseguire il seguente comando a partire dalla root directory del progetto:

- java -cp ./out/production:lib/jackson-annotations-2.9.7.jar:lib/jackson-core-2.9.7.jar:lib/jackson-databind-2.9.7.jar winsome.WinServerMain

E' presente un semplice script ./startServer.sh per la compilazione ed esecuzione automatica del server
Il file .jar associato è presente nella cartella out/production/artifacts. Per eseguirlo basta inserire il seguente comando a partire dalla root directory del progetto:

- java -jar out/artifacts/WinServer.jar

5.2 Interfaccia da linea di comando

Per compilare l' interfaccia da linea di comando è necessario eseguire il seguente comando a partire dalla root directory del progetto:

- javac -cp ./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar src/winsome/WinCLIClientMain.java src/winsome/client/CLI/*.java src/winsome/client/*.java src/winsome/jsonUtility/*.java src/winsome/resourseRappresentation/*.java src/winsome/RESTfulUtility/*.java src/winsome/server/*.java -d out/production

In questo modo compiliamo tutti i file .java necessari, includiamo le librerie Jackson utilizzate per formattare i dati in formato JSON e indirizziamo tutto l'output prodotto nella cartella out/production
Per eseguire l' interfaccia da linea di comando è necessario eseguire il seguente comando a partire dalla root directory del progetto:

- java -cp ./out/production:./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar winsome.WinCLIClientMain

E' presente un semplice script ./startCLIClient.sh per la compilazione ed esecuzione automatica dell'interfaccia da linea di comando.

Il file .jar associato è presente nella cartella out/production/artifacts. Per eseguirlo basta inserire il seguente comando a partire dalla root directory del progetto:

- java -jar out/artifacts/WinCLIClient.jar

5.3 Interfaccia grafica

NOTA:
La compilazione e l'esecuzione dell'interfaccia grafica sono platform-specific: le regole che seguono sono valide nel mio computer con un' architettura x64 e sistema operativo Linux Ubuntu. In caso di warning in fase di esecuzione dei file compilati è necessario scaricare la corretta versione della libreria openjfx dal sito**: https://gluonhq.com/products/javafx/** a seconda del sistema operativo e dell'architettura del computer su cui si esegue l'applicazione.
In seguito è necessario copiare il file nella cartella lib e sostituire al comando export che seguirà nelle istruzioni il path alla cartella lib presente nel file scaricato.
L'operazione potrebbe risultare noiosa ma è importante visualizzare l'interfaccia grafica per vivere a pieno l'esperienza WINSOME. In ogni caso, se per mancanza di tempo tali operazioni non siano effettuabili, nella relazione sono presenti numerosi screenshot della GUI realizzata.

Per compilare l' interfaccia grafica è necessario eseguire i seguenti comandi:

- export PATH_TO_FX=./lib/openjfx-17.0.1_linux-x64_bin-sdk/javafx-sdk-17.0.1/lib
- javac --module-path $PATH_TO_FX --add-modules javafx.controls,javafx.fxml -cp ./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar src/winsome/WinGUIClientMain.java src/winsome/client/GUI/*.java src/winsome/client/*.java src/winsome/jsonUtility/*.java src/winsome/resourseRappresentation/*.java src/winsome/RESTfulUtility/*.java src/winsome/server/*.java -d out/production

Per eseguire l'interfaccia grafica è necessario eseguire i seguenti comandi:

- java --module-path $PATH_TO_FX --add-modules javafx.controls,javafx.fxml -cp ./out/production:./lib/jackson-annotations-2.9.7.jar:./lib/jackson-core-2.9.7.jar:./lib/jackson-databind-2.9.7.jar winsome.WinGUIClientMain

(se si scarica una versione diversa della libreria javaFX sostituire al comando export PATH_TO_FX= l'indirizzo della cartella lib all'interno della libreria scaricata)

E' presente un semplice script ./startGUIClient.sh per la compilazione ed esecuzione automatica dell'interfaccia grafica.

Il file .jar associato è presente nella cartella out/production/artifacts. Per eseguirlo basta inserire I seguenti comandi:

- export PATH_TO_FX=./lib/openjfx-17.0.1_linux-x64_bin-sdk/javafx-sdk-17.0.1/lib
- java --module-path $PATH_TO_FX --add-modules javafx.controls,javafx.fxml -jar out/artifacts/WinGUIClient.jar