Winsome — progetto di laboratorio di reti

Samuele Bonini

10 gennaio 2022

Indice

1 Note sulla compilazione ed esecuzione

Per compilare il progetto, **si esegua lo script** `compile.sh`, collocato all'interno della root del progetto. Lo script reperisce i file `.java` necessari alla compilazione dalle *classh paths* e utilizza il comando `javac` per creare la build, includendo anche i file `.jar` delle librerie esterne utilizzate (fonte).

Per eseguire la classe `MainClient`, si esegua lo script `client.sh`, mentre per la classe `MainServer` (che deve essere eseguita prima del client), si esegua lo script `server.sh`. Entrambe le classi possono altresì essere utilizzate via i rispettivi file `jar`, entrambi collocati nella root directory.

2 Architettura di massima del progetto

2.1 Suddivisione in layer

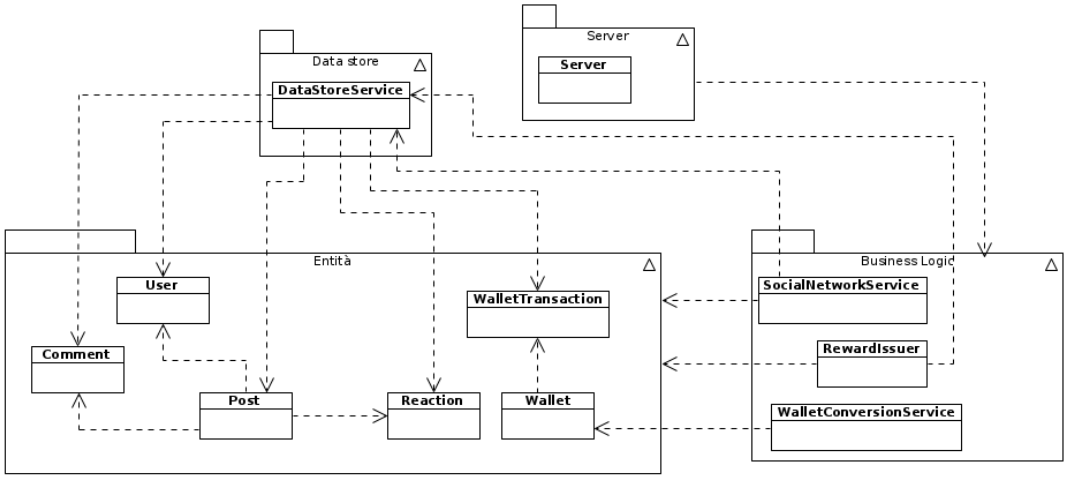
Nella realizzazione del software in oggetto, si è scelto di utilizzare un'architettura a strati, suddividendo le classi in gioco all'interno di insiemi disgiunti in base alle responsabilità e alle dipendenze. Lo scopo di utilizzare layer ben distinti è quello di ottenere un basso grado di accoppiamento tra le componenti del sistema, mantenendo altresì una buona *separation of concerns*. Avere un basso grado di accoppiamento tra le classi implica la possibilità di sviluppare in maniera quanto più indipendente le varie componenti, modificandole (o anche riscrivendole da zero) all'occorrenza, oltre che avere un vantaggio nelle fasi di testing e debugging, ove è più facile isolare eventuali fault.

In seguito all'analisi dei requisiti, in fase di progettazione sono stati individuati i seguenti layer:

- **Entità**: sono le classi che rappresentano il dominio dell'applicazione e incarnano l'universo delineato dal progetto
- **Data store**: espone le entità al mondo esterno attraverso un'interfaccia controllata e definisce le operazioni primitive sulle stesse
- **Business logic**: implementa operazioni più complesse servendosi dalle primitive esposte dallo store e, talvolta, dalle entità stesse; aggiunge controlli di vario tipo (permessi, *well-formedness* delle richieste, ecc.); fornisce all'esterno un'API più ricca per interagire con le entità
- **Server**: gestisce la connessione logica coi client, espone un'API per accedere ai layer sottostanti secondo le regole di business, si occupa della formattazione e presentazione dei dati in entrata e in uscita

La regola generale è che le classi di un layer hanno dipendenze solo da classi dello stesso layer o di layer più interni (*inward dependency principle*). Si noti infine che i nomi dei layer sopracitati non sono in corrispondenza biunivoca coi package Java utilizzati nel progetto, i quali sono stati identificati con un criterio differente e più legato ai dettagli implementativi.

Il seguente diagramma delle dipendenze esemplifica il principio di *inward dependency* applicato alle classi di questo progetto. Si noti che le dipendenze sono indotte solo tra classi appartenenti allo stesso layer o verso classi dei layer più interni.



Questo documento è strutturato in maniera tale da illustrare, come prima cosa, le caratteristiche delle entità di dominio e le operazioni primitive a esse associate, per poi spostarsi progressivamente sui layer più esterni dell'architettura, dando un'idea di come le classi appartenenti allo strato della business logic sono implementate, chiarendo infine come viene realizzata, agli atti, la comunicazione tra il server e il client.

2.2 Protocollo applicazione

Il protocollo utilizzato a livello applicativo è HTTP. Il server espone un'API di tipo REST misto a RPC. Le richieste e le risposte HTTP vengono incarnate rispettivamente dalle classi `RestRequest` e `RestResponse`, che dispongono di metodi per convertirle da (risp. in) stringhe, consentendone la (de)serializzazione e lettura (risp. scrittura) mediante socket.

L'autenticazione utilizza il protocollo **bearer token**, schema definito nell'RFC 6750. Il protcollo prevede che le richieste vengano autenticate inserendo uno header (`Authorization`) contenente un token di 128 bit. La classe `AuthenticationMiddleware` (illustrata più in dettaglio nel seguito), responsabile della verifica delle credenziali nelle richieste HTTP, controlla la presenza e la validità di questo header.

Il token viene ottenuto per la prima volta dal client in fase di login: la password fornita dall'utente all'atto della registrazione, che viene mantenuta nel database criptata utilizzando l'algoritmo MD5, viene confrontata con quella inviata durante il login (dopo aver cifrato anch'essa); in caso di esito positivo, il server emette un token che l'utente utilizzerà nelle successive richieste (fino al logout).

3 Le entità di dominio

3.1 Gli utenti

La classe `User` rappresenta un utente registrato a Winsome. A esso è associato uno username, una password (criptata come descritto nel capitolo precedente) e un `Set` di tag, rappresentati da stringhe *lowercase*. L'univocità dello username non è gestita a livello entità ma nello store.

3.2 Post, commenti, reazioni

La classe `Post` rappresenta un post all'interno di Winsome. Contiene informazioni circa il titolo, il contenuto e il timestamp di creazione. L'identificativo del post è uno UUID.

Il post contiene una stringa che si riferisce allo username del suo autore. La scelta di utilizzare una stringa anziché un riferimento a un oggetto `User` deriva dalla volontà di gestire l'integrità referen-

ziale a livello dello store anziché all'interno delle entità stesse: si è scelto, in altre parole, di non utilizzare l'**active record pattern**, in favore della presenza di un layer dedicato all'accesso ai dati.

I commenti e le reazioni (upvote o downvote) ai post vengono aggregati all'interno di due insiemi presenti nella classe. La scelta di utilizzare un `TreeSet` per entrambi deriva dalla necessità di mantenere l'ordine temporale di inserimento.

Infine, i post contengono opzionalmente un riferimento a un altro post: ciò viene utilizzato per implementare la feature di **rewin**. Questo permette, in principio (anche se non presente in quanto fuori dalla specifica), di effettuare il *rewinning* di un post aggiungendo anche un testo come contenuto del post, come un "commento in evidenza" (feature presente, per esempio, nei retweet, a cui il rewin si ispira).

3.3 Il portafoglio

La classe `Wallet` rappresenta il portafoglio di un utente registrato a Winsome. Al momento della registrazione di un utente, un'istanza di questa classe viene creata e associata al nuovo `User` all'interno del data store (si veda il relativo capitolo).

Al suo interno, è definita la *inner class* `WalletTransaction`, che rappresenta un cambiamento nel bilancio del portafoglio. Il metodo `getBalance` permette di ottenere il bilancio aggregando l'ammontare di tutte le transazioni associate al portafoglio.

3.4 Serializer

La classe `Serializer<T>` è responsabile delle operazioni di serializzazione e de-serializzazione delle entità. L'utilizzo del polimorfismo parametrico permette di (de)serializzare tutte le classi che lo necessitano con un singolo metodo per operazione, senza ri-definirne la logica.

Internamente, la classe utilizza la libreria Jackson e l'`ObjectMapper`. Le entità da (de)serializzare sono annotate, all'occorrenza, con le *annotations* del pacchetto `jackson.annotation` per avere un controllo più fine su quali proprietà e campi vengono serializzati.

4 Il data store

La classe `DataStoreService` implementa il data access layer. Essa costituisce un'astrazione posta sopra al sistema di storage sottostante, qualunque esso sia (in questo caso, come richiesto dalla specifica, si utilizza il filesystem e file in formato JSON, ma la presenza di questo layer permetterebbe, se necessario, di cambiare lo storage system utilizzando, per esempio, un DBMS, senza alterare il resto del progetto).

Lo store gestisce gran parte della concorrenza all'interno del progetto: incapsulando l'accesso concorrente ai dati in una sola classe, è possibile sviluppare il layer della business logic in maniera prevalentemente **thread-unaware**, il che semplifica fortemente l'implementazione e suddivide meglio le responsabilità.

4.1 Strutture dati e concorrenza

La scelta delle strutture dati segue il criterio di **indicizzazione** dei dati per consentire una migliore performance nei lookup e nelle scritture. Questo significa che la *ratio* dietro la presenza di strutture dati che apprentemente portano a ridondanza (e talvolta a una riduzione del grado di normalizzazione dei dati) è quella di ottimizzare le operazioni sui dati più comuni.

Lo strumento principale per gestire la concorrenza è l'utilizzo della `ConcurrentHashMap` di Java. L'utilizzo delle varianti del metodo `compute` della struttura in questione garantisce l'atomicità delle operazioni e pertanto la consitenza dei dati. In pochi casi (come la cancellazione di un post, in

seguito alla quale è necessario cancellare anche eventuali rewin del post), viene ammessa una breve finestra all'interno della quale la consistenza è compromessa: questo pattern è conosciuto come *eventual consistency* e trova ragion d'essere nell'obiettivo di compromettere la performance il meno possibile, specialmente in operazioni dove la temporanea perdita di consistenza non appare avere effetti negativi rilevanti sul modo in cui l'utente percepisce lo stato del sistema.

All'interno dello store si trovano le seguenti mappe:

- `users`: mappa username su istanze della classe `User`; è il metodo principale per reperire un utente
- `userPosts`: mappa username su insiemi di post (viene utilizzato un `TreeSet` per garantire l'ordinamento cronologico consistente); utilizzata per avere un accesso rapido ai post quando vengono cercati per autore
- `sessions`: mappa token di autenticazione su utenti; utilizzata per autenticare gli utenti in base al valore del campo header `Authorization`
- `posts`: mappa UUID su istanze di `Post`; utilizzata per cercare post per ID
- `followers`: mappa username su insiemi di username; mantiene le relazioni di follower tra gli utenti
- `wallets`: mappa username su portafogli (si veda il paragrafo sulla classe `Wallet`)
- `notificationCallbacks`: mappa username su `IClientFollowerNotificationService`; utilizzata per reperire l'istanza del client per il servizio di notifica dei follower via RMI callback

4.2 Persistenza dello stato

Al momento dell'istanziazione, la classe `DataStoreService` inizializza un thread che, periodicamente, effettua il salvataggio dello stato dello store. Il salvataggio avviene mediante una serializzazione dell'istanza dello store e successiva scrittura su un file indicato al momento della creazione dell'istanza. La classe espone altresì il metodo statico `restoreOrCreate` che, passato il nome di un file JSON, tenta di ripristinare lo stato a partire da quel file, de-serializzandolo e, in caso di impossibilità di completare l'operazione (file inesistente o malformato), restituisce una nuova istanza dello store.

5 Il business logic layer

Lo strato dei servizi/business logic fornisce all'esterno un'interfaccia mediante la quale implementa tutte le operazioni che devono essere esposte all'utente.

5.1 La classe SocialNetworkService

La classe `SocialNetworkService` implementa le operazioni associate alle route dell'API REST/RPC. I metodi handler presenti in questa classe sono in corrispondenza biunivoca con i path dell'API (si veda capitolo sul router).

In generale, uno handler è una funzione che prende come parametro un oggetto di tipo `RestRequest` e restituisce una `RestResponse`. Al suo interno, ha luogo la logica di business come la validazione dei parametri della richiesta, il controllo dei permessi, il *dispatching* delle azioni richieste verso il data store, e la serializzazione dei dati in uscita per la costruzione della risposta.

Gli handler restituiscono direttamente una `RestResponse` solo nel caso in cui l'esecuzione abbia successo. Se si verifica un errore, i metodi in questione non restituiscono una risposta (per esempio `4xx`), bensì sollevano un'eccezione. La gestione degli errori in fase di *handling* di una richiesta viene

così demandata al server (come illustrato nel relativo capitolo), il quale invierà una risposta HTTP contente l'errore appropriato.

Questa strategia, ispirata al modo in cui il framework Django gestisce gli errori nelle sue *views*, serve ad astrarre ulteriormente i casi in cui la richiesta non ha successo: anziché avere il codice per la gestione degli errori sparpagliato (e duplicato) tra i vari handler, essi si limitano a lanciare un'eccezione che viene catturata e gestita in una singola locazione. Così diventa facile, per esempio, aggiungere qualsiasi tipo di logica *custom* per la gestione degli errori. Si pensi per esempio all'aggiunta di un sistema di logging: basterebbe chiamare il metodo esposto dal servizio nel singolo spot in cui viene catturata l'eccezione.

La classe `SocialNetworkService` è prettamente *stateless*: le chiamate ai metodi al suo interno causano cambiamenti nello stato gestito dallo store, ma la classe in sé non contiene stato interno. Questo permette, in principio, sia di utilizzare la classe come un *singleton* (come viene effettivamente fatto), oppure di scalarla orizzontalmente, dislocando più istanze di essa su diversi nodi worker (nel caso di un sistema distribuito), a patto di mantenere un riferimento allo store centralizzato.

5.2 La classe RewardIssuer

La classe `RewardIssuer` realizza il servizio di ricompense di Winsome. Implementa l'interfaccia `Runnable` e una singola istanza di essa viene creata dal server al momento dell'avvio ed eseguita in un thread. Nel suo metodo `run`, la classe entra in un ciclo infinito, all'interno del quale attende via `Thread.sleep` il tempo stabilito nel file di configurazione del server, per poi chiamare un metodo che calcola le nuove ricompense per tutti gli utenti.

Il calcolo sui singoli post viene effettuato secondo la formula data dalla specifica. Per reperire i dati necessari al calcolo, il reward issuer si serve di una classe *helper* interna, `PostRewardData`: in essa, vengono aggregati i dati necessari all'applicazione della formula a un singolo post, nonché al corretto assegnamento delle ricompense ai curatori. In particolare, la classe contiene l'insieme di: utenti che hanno messo un *upvote* dall'ultimo calcolo delle ricompense, utenti che hanno commentato dall'ultima iterazione, nuovi commenti e nuove reazioni.

Una volta completato il calcolo delle ricompense, viene aggiornato un campo nella classe che tiene traccia del timestamp dell'ultimo calcolo ricompense e viene inviata una notifica multicast mediante un `DatagramSocket`.

5.3 La classe WalletConversionService

La classe `WalletConversionService` è responsabile per la simulazione del cambio nel tasso di conversione tra Wincoin e Bitcoin. Per ottenere un tasso casuale, viene effettuata una richiesta HTTP all'API di random.org. La richiesta è autenticata mediante l'utilizzo di una `apiKey`, come spiegato nella documentazione dell'API di random.org.
La connessione al servizio viene aperta utilizzando la classe `HttpURLConnection`. La richiesta viene scritta utilizzando un `writer` di tipo `DataOutputStream` e la successiva risposta viene letta con un `BufferedReader` e passata a uno `StringBuilder`. La stringa risultante viene *parsata* per verificare che l'interazione abbia avuto successo ed estrarre il valore casuale restituito dal servizio remoto.

Dato che il servizio remoto potrebbe, in generale, non essere disponibile, la classe è dotata di un **circuit breaker** allo scopo di migliorare la disponibilità del servizio. L'idea è la seguente: quando una richiesta ha successo, il valore restituito da random.org viene salvato in una variabile interna che svolge la funzione di cache.
Nel caso in cui una successiva richiesta fallisca, il servizio proverà periodicamente a ricontattare il server remoto fino a raggiungere il numero `MAX_RETRIES` di tentativi. A quel punto, il circuit breaker verrà attivato. L'effetto risultante è che il valore in cache verrà restituito e, per un periodo di tempo pari a `CIRCUIT_BREAKER_COOL_DOWN` secondi, tutte le richieste al servizio restituiranno immediatamente il valore in cache, senza provare a contattare l'API di random.org.

L'impiego di questo pattern, comune in architetture a microservizi e sistemi distribuiti in genere, serve a minimizzare il numero di richieste inviate durante un periodo in cui il servizio remoto è, con

alta probabilità (dato che le richieste recenti hanno dato esito negativo), ancora non raggiungibile.

La classe viene utilizzata come *singleton* ma, a differenza di `SocialNetworkService`, contiene stato interno (il valore *cached* e i dati del circuit breaker). Per permettere la condivisione fra thread della singola istanza di questa classe, le parti di codice dove lo stato interno viene acceduto sono protette da **monitor** (mediante il costrutto `synchronized`).

5.4 I servizi di registrazione utente e notifica

Il servizio di registrazione utente è esposto dalla classe `UserRegistrationService`, che a sua volta implementa `UserRegistrationInterface`. Il singolo metodo a disposizione, `registerUser`, prende in input i dati necessari per la registrazione (username, password e insieme di tag), ne verifica la correttezza e, in caso di esito positivo, provvede alla creazione e restituzione di un nuovo oggetto `User`. Gli appropriati errori vengono restituiti in caso di parametri invalidi oppure se lo username scelto è già in uso.

Il sistema di notifiche per i nuovi follower è implementato, come da specifica, utilizzando il meccanismo RMI callback. La classe `FollowerNotificationService` espone il metodo `subscribe`, chiamato dal client al momento del login. La classe dispone altresì del metodo `notifyUser`, che viene chiamato negli handler `followUser` e `unfollowUser` al momento del completamento dell'operazione. Il metodo in questione reperisce un riferimento a `IClientFollowerNotificationService`, acquisito al momento del login e associato nello store allo username dell'utente, e chiama il metodo `updateFollowerList` esposto da quest'ultima interfaccia, passando come parametro il nuovo insieme di follower. Il client provvede quindi ad aggiornare la propria lista locale utilizzando il valore fornito dal servizio.

6 L'API e il router

La gestione delle richieste da parte del server segue un pattern simile al **model-view-controller** (MVC). I *model* sono rappresentati in maniera naturale dalle entità di dominio e gli handler della classe `SocialNetworkService` costituiscono le *views*. Il router rappresenta il *controller*, ovvero il componente che mappa le coppie $< path, metodo >$ su handler dell'API.

6.1 La classe ApiRoute

La classe `ApiRoute` rappresenta una route dell'API e i metodi handler a essa associati. Contiene un campo stringa rappresentante il path e una `Map` che associa metodi HTTP a stringhe che rappresentano nomi di metodi (spiegato più in dettaglio nel seguito).

6.2 La classe ApiRouter

All'avvio del server, viene istanziato un `ApiRouter`, il quale carica in memoria degli oggetti di tipo `ApiRoute`, de-serializzandoli da un file JSON.

La responsabilità del router è quella di risolvere i path richiesti restituendo un riferimento all'appropriato metodo handler per la richiesta. Contiene un metodo principale, `getRequestHandler`, che prende in ingresso una `RestRequest`. Il metodo si occupa di estrarre il path e il metodo HTTP della richiesta e di cercare un match all'interno del suo insieme di route.

Se si ha una corrispondenza (ovvero è associato il nome di un metodo handler alla coppia $< path, metodo >$), viene restituito il riferimento a un metodo della classe `SocialNetworkService`, che verrà poi invocato coi corretti parametri per soddisfare la richiesta (si veda capitolo successivo per il flusso completo).

Il meccanismo appena descritto utilizza la feature della *reflection* e la classe Java `Method`, collocata nel pacchetto `java.lang.reflect`.

Si è scelto di utilizzare questo approccio (anziché, per esempio, un controllo `switch` sul path della richiesta) per definire l'API in maniera **dichiarativa** (e non imperativa): avere la definizione delle route in un file JSON e non nel codice stesso permette di ottenere un basso grado di accoppiamento tra la logica e la descrizione dell'interfaccia; si pensi per esempio all'aggiunta di un nuovo *entry point* nell'API: se la descrizione dell'interfaccia fosse contenuta nel codice, la modifica allo schema richiederebbe anche una modifica al codice del router, mentre con questo tipo di definizione è sufficiente aggiornare il file JSON contenente lo schema dell'API (dopo aver, eventualmente, implementato il nuovo handler).

7 Il server

7.1 Funzionamento e gestione delle connessioni

Il modello utilizzato per l'implementazione del server è il multiplexing dei canali via `Selector` NIO con l'impiego una threadpool per la gestione delle richieste. All'avvio il server, dopo aver aperto un `ServerSocketChannel` e aver esportato gli stub necessari per l'esposizione dei servizi basati su RMI, entra in un ciclo infinito.

All'interno del ciclo, il server utilizza un `Selector` per accettare nuove connessioni e richieste. Quando un client diventa *readable*, viene chiamato sulla key corrispondente il metodo `readFromKey`. Al suo interno, viene letta la richiesta inviata dal client, la quale viene poi gestita secondo il flusso descritto nel paragrafo *Gestione di una richiesta HTTP*.

Il task della gestione della richiesta viene sottomesso a un `ExecutorService`, permettendo al thread principale di continuare ad accettare richieste mentre gli worker gestiscono quelle già prese in carico. Per la sottomissione delle richieste al pool viene utilizzata la classe `CompletableFuture`.

Per prima cosa, il metodo `handleRequest` (che si occupa materialmente della gestione della richiesta) viene passato come callback al metodo `supplyAsync` della classe `CompletableFuture`. Questo fa sì che il task venga sottomesso al pool `ForkJoinPool.commonPool`, che è gestito internamente dalla JVM, inizializzato staticamente, ed è ottimizzato con la tecnica del *work stealing* per massimizzare l'efficienza e l'utilizzo dei thread. Al completamento del task da parte di un worker, viene chiamato il callback passato al metodo `thenAccept`: questa funzione lambda prende come argomento la `RestResponse` restituita dalla gestione della richiesta e la salva nell'*attachment* associato al client. Dopodiché, le `interestOps` della key del client vengono impostate in modo da tenere traccia di quando il client diventerà *writable*.

Una volta verificatosi questo evento, il metodo `writeToKey` viene chiamato e causa la scrittura della risposta precedentemente generata sul socket associato al client. Al termine della scrittura, le `interestOps` della key in questione vengono nuovamente settate in modo da tracciare la *readability*.

7.2 Gestione di una richiesta HTTP

Dopo aver letto dal socket e ricostruito una `RestRequest` inviata da un client, il flusso di gestione della richiesta da parte del server si compone dei seguenti passi:

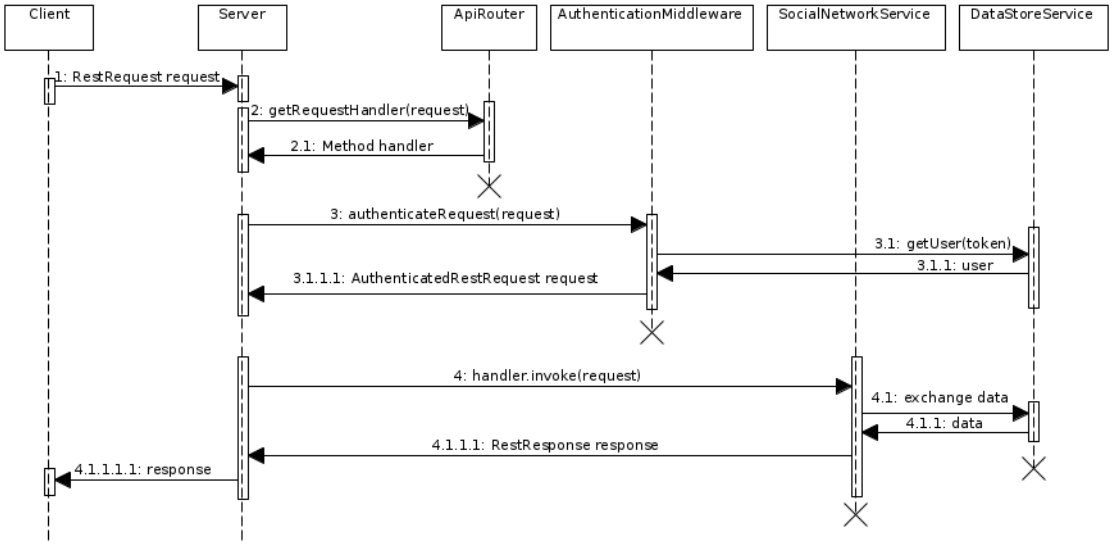
1. Il path e metodo della richiesta vengono utilizzati per cercare una corrispondenza nel router.
   1.1. Se il path non dà corrispondenza, viene restituito l'errore 404. Se il metodo non dà corrispondenza, viene restituito l'errore 405. Se il lookup ha successo, viene reperito il riferimento al metodo handler appropriato.
2. L'`AuthenticationMiddleware` analizza gli header della richiesta alla ricerca del token nel campo `Authorization`. Questo controllo viene omesso in fase di login.
   2.1. Se il token non è presente, viene restituito l'errore 401. Se il token è invalido, viene restituito l'errore 400. Se la verifica ha successo, la richiesta viene autenticata (incapsulandola in una `AuthenticatedRestRequest`) e associata all'utente corrispondente al token.

3. Lo handler restituito al punto 1 viene invocato passandogli la richiesta autenticata del punto 2.

   3.1. Se si verifica un errore all'interno dello handler, l'eccezione viene catturata come `InvocationTargetException`. La tecnica di *exception chaining* permette di estrarre l'eccezione originale sollevata dallo handler e inviare in risposta l'errore appropriato. Qualsiasi altro tipo di eccezione sollevata non direttamente dallo handler causa l'emissione di un errore 500. Se l'esecuzione ha successo, lo handler restituisce una `RestResponse`.

4. La risposta viene salvata nell'*attachment* associato al client e verrà scritta sul socket corrispondente quando il client diventa *writable*.

Il seguente diagramma di sequenza esemplifica lo schema delle interazioni tra classi sintetizzato sopra.



7.3 La classe ServerConfig

La classe `ServerConfig` rappresenta l'insieme delle impostazioni del server che, da specifica, devono poter essere regolate. Il file di configurazione risulta quindi essere un file JSON contenente un'istanza serializzata di questa classe. Il vantaggio principale derivante dall'uso di questo approccio è il riutilizzo delle componenti già esistenti all'interno del progetto: per leggere il file di configurazione basta, infatti, istanziare un `Serializer<ServerConfig>` e de-serializzare il contenuto del file specificato.

Di default, il server utilizzerà il file `config.json` all'avvio, ma è possibile sovrascrivere questa impostazione eseguendo la classe `MainServer` con un argomento da linea di comando, per esempio `java MainServer myconfig.json`.

Al fine di permettere di avviare il server con il minimo livello di configurazione manuale possibile, la classe `ServerConfig` è dotata di valori di default per ciascun *setting*, che verranno utilizzati se omessi nel file di configurazione scelto.

Il client, per semplicità, ha accesso a un riferimento a `ServerConfig` in maniera analoga al server: questo permette di reperire facilmente informazioni come l'indirizzo IP e la porta per avviare la connessione TCP. Nell'ipotesi di un sistema realmente distribuito, dove client e server non risiedono sulla stessa macchina, si utilizzerebbe un sistema più sofisticato per il reperimento di tali dati, come per esempio l'accesso a un terzo servizio (se il server fosse dotato di un nome di dominio, per esempio, basterebbe un lookup DNS). Si noti che, seppure la maggior parte dei dati vengono letti dal client direttamente dall'istanza di `ServerConfig`, questo non avviene per gli estremi del gruppo multicast, in quanto la specifica richiede che sia il server a comunicarli: essi vengono infatti inclusi nel corpo della risposta alle richieste di login.

8 Il client CLI

All'avvio, il client reperisce utilizzando l'RMI `Registry` i servizi di registrazione utente e notifica per i follower, precedentemente esportati dal server. Dopodiché, viene aperta una connessione TCP utilizzando un `SocketChannel` NIO. Una volta stabilita la connessione al server, il client entra in un ciclo infinito dove attende di leggere istruzioni da `stdin`. Ogni istruzione letta in input viene divisa in token in corrispondenza del carattere spazio: il primo token (o i primi due, per le istruzioni composte da due parole) viene utilizzato per cercare corrispondenza con uno dei comandi supportati. Successivamente, vengono utilizzati tre metodi – `getStringArgument`, `getIntArgument` e `getUUIDArgument`, a seconda del contesto – per acquisire i parameteri del comando ricevuto, verificandone l'arietà e il tipo. Infine, se tutti i controlli hanno esito positivo, viene chiamato il metodo handler corrispondente al comando con i parametri dati dall'utente.

Dopo aver effettuato il login, il client ottiene gli estremi (indirizzo IP e porta) per unirsi al gruppo multicast per le notifiche sugli aggiornamenti dei portafogli, inviatigli dal server all'interno della risposta alla richiesta di autenticazione, e vi si connette in un thread separato. Il token di autenticazione altresì contenuto nella risposta viene salvato in una variabile locale e utilizzato per autenticare le richieste future. Infine, una volta ottenuto il token, il client si registra via RMI callback al servizio di notifica per i follower. Il token viene passato come argomento al metodo per la registrazione, certificando così l'identità dell'utente.

8.1 Gestione di una richiesta al server

Gli handler del client implementano il seguente schema:

1. Viene generata una `RestRequest` coi parametri (path, header e body) appropriati in base all'operazione
2. Viene chiamato il metodo `receiveResponse`, che wrappa la richiesta in un `ByteBuffer` e la scrive sul socket che mantiene la connessione al server
3. Viene letta la risposta. Se la risposta contiene un errore (codice `4xx` o `5xx`), viene sollevata una `ClientOperationFailedException`, altrimenti viene restituita la risposta allo handler chiamante
4. All'interno dello handler, viene processato il contenuto della risposta. Il *processing* effettuato dipende dalla semantica dell'operazione; nella maggior parte dei casi, viene istanziato un `Serializer` che ricostruisce un'entità (o un array di entità) e lo restituisce in uscita.

La classe `ClientOperationFailedException` viene utilizzata per rappresentare il fallimento di un'interazione col server, dovuto nella maggior parte dei casi alla ricezione di una risposta HTTP contenente un codice di errore. Quando un'eccezione di questo tipo viene lanciata, possono esserle passati come parametro la richiesta e la risposta che costituiscono l'interazione fallita. Questi dati vengono utilizzati dal client per stampare il messaggio di errore appropriato (si veda la sezione *Gestione dei messaggi*).

8.2 Rendering dei dati

La maggior parte delle operazioni svolte dal client si conclude con la stampa su `stdout` di dati ricevuti dal server. Per astrarre la logica di ricezione dei dati da quella legata alla formattazione (o *rendering*) degli stessi, si è utilizzata un'interfaccia parametrica dedicata: `IRenderer<E>`.

Le classi che implementano quest'interfaccia dispongono del metodo `render`, *overloaded* in modo da poter essere utilizzato sia con singole istanze che con array. Il metodo `render` si occupa, preso in input un oggetto del tipo assegnato dalla classe che implementa l'interfaccia, di formattarlo e restituire una stringa che rappresenta una versione *pretty-printed* dell'oggetto.

Il client, all'atto della ricezione di dati dal server, istanzia un `Renderer` del tipo appropriato. Al termine dell'interazione, la stringa restituita dal renderer viene stampata su `stdout`.

8.3 Gestione dei messaggi

L'approccio scelto per la gestione dei messaggi (sia informativi che di errore) è di mantenerli in strutture dati separate dal resto del codice. L'idea è quella di associare a ogni messaggio un codice mnemomico, e di accedere ai messaggi con un lookup nella struttura dedicata utilizzando il codice corrispondente.

L'utilizzo di questo metodo è motivato dall'obiettivo di disaccoppiare la logica del client dalla presentazione dei dati (non è buona pratica, in generale, avere *magic strings* all'interno del codice). Inoltre, come possibile estensione, si consideri cosa succederebbe se si volesse tradurre il client in più lingue. Disaccoppiando i messaggi dal codice dei vari metodi e impiegando un tool di traduzione, quale `i18n`, basterebbe modificare l'accesso alla struttura dati che contiene i messaggi testuali; diversamente, se le stringhe fossero *hard-coded*, bisognerebbe intervenire sul codice modificandole *una ad una*.

La maggior parte dei messaggi del client sono salvati all'interno della mappa `clientMessages`. Essa viene inizializzata e resa immutabile utilizzando la *factory* `Collections.unmodifiableMap`.

Per avere un controllo più fine sui messaggi di errore, una seconda struttura viene impiegata: `outcomeMessages`. Essa mappa nomi di path su mappe, che a loro volta mappano interi (rappresentanti codici HTTP di esito) su stringhe. L'idea è di fornire messaggi contestualizzati all'operazione e all'esito risultante. La struttura in questione, popolata nello stesso modo della precedente, viene acceduta quando si verifica una `ClientOperationFailedException`, se essa non è già stata inizializzata con un messaggio: la `RestRequest` e `RestResponse` contenute nell'eccezione vengono utilizzate per il lookup nella mappa.

9 Il client GUI

Il client GUI è costituito da una web app realizzata utilizzando HTML5 e *vanilla* JavaScript. È stato utilizzato il framework Tailwind CSS per avere un set di classi di stile predefinite sopra le quali costruire il layout.

La motivazione dietro la realizzazione di un client web è di esemplificare il funzionamento del server anche con client eterogenei (in particolare, non scritti in Java), mostrando che l'API esposta è *compliant* con le specifiche REST/RPC e può essere consumata anche da browser.

Per avviare la web app, è sufficiente aprire con un browser il file `index.html` presente all'interno della cartella `guiClient`, nella root del progetto.

9.1 Realizzazione

La struttura della web app è quella di *single-page application* (SPA): tutte le funzionalità sono raggruppate all'interno di una singola pagina HTML, e l'utilizzo di JavaScript permette di mostrare programmaticamente solo la parte di pagina inerente all'operazione che si sta svolgendo.

Le richieste HTTP sono effettuate utilizzando la libreria **Axios**.

Il token di autenticazione ricevuto in fase di login viene salvato nei *common headers* di Axios e automaticamente incluso nelle richieste effettuate dopo il login. L'applicazione è dotata di un sistema *ricordami*, che permette di aggiornare la pagina o chiudere e riaprire il browser rimanendo loggati. La funzionalità è implementata utilizzando il `localStorage` di JavaScript: i dati vengono salvati nello storage del browser (in maniera simile ai cookie) e ripristinati una volta riaperta la pagina web.

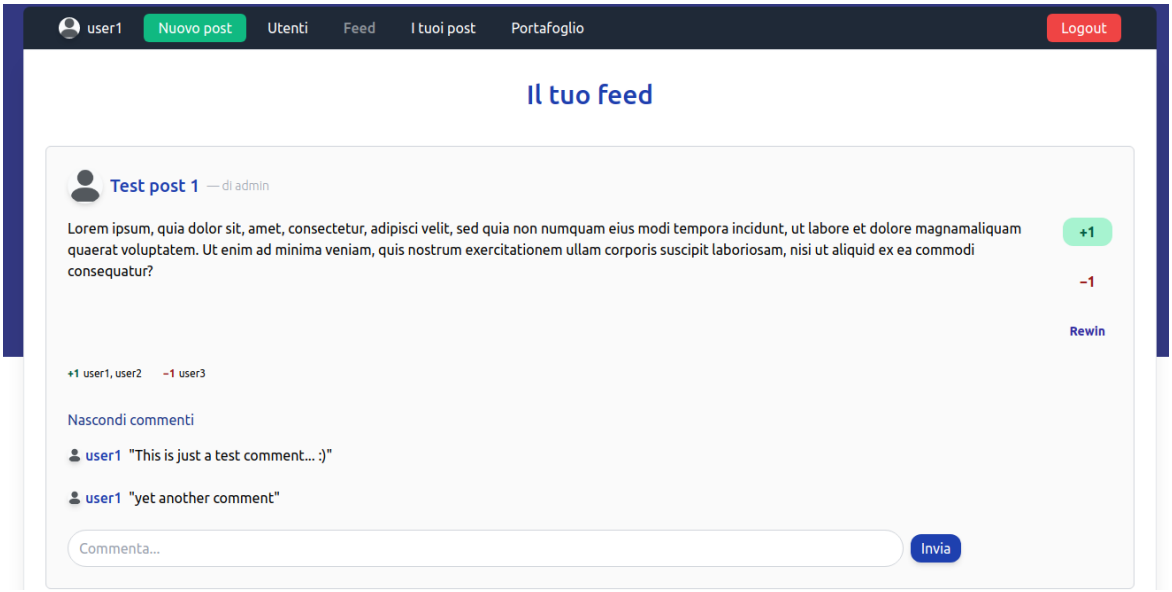
9.2 Limitazioni

Il client web implementa tutte e sole le funzionalità interagibili tramite l'API REST/RPC del server. In altre parole, il sistema di registrazione utente, le notifiche per i follower e le notifiche multicast

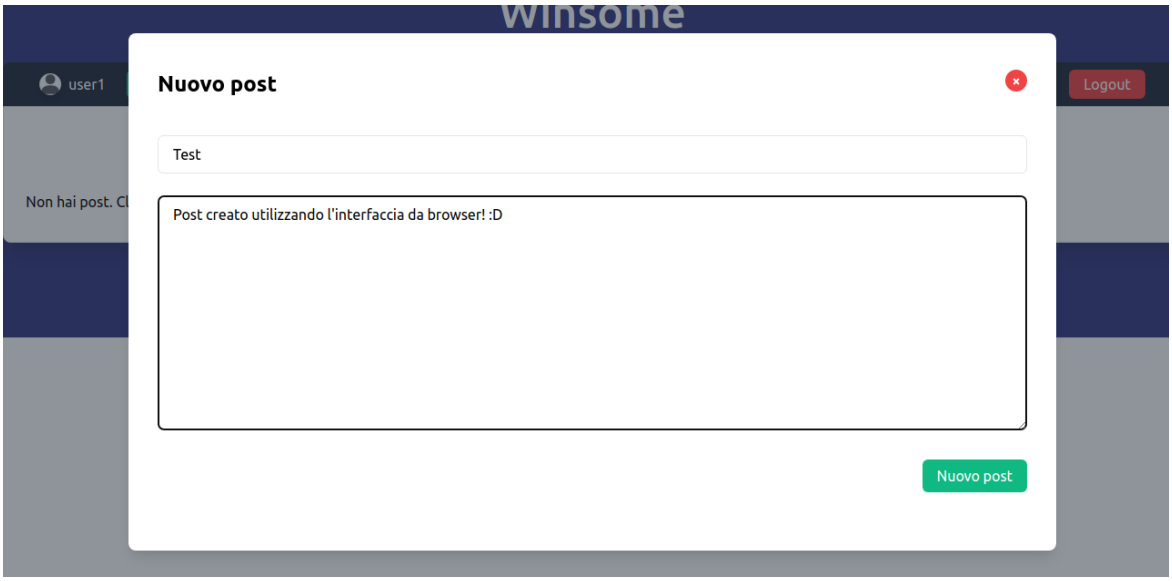
non sono presenti in questa versione del client.

Per ovviare all'impossibilità di registrarsi, si può registrare un utente utilizzando il client CLI ed effettuare il login dal client web, oppure utilizzare l'utente che viene **fornito di default** (incluso per permettere di testare agilmente la web app senza dover utilizzare prima il client CLI): è sempre possibile, infatti, effettuare il login utilizzando lo username `admin` e la password `pass`.

9.3 Alcuni screenshot dell'utilizzo della GUI




Winsome
admin
Nuovo post
Utenti
Feed
I tuoi post
Portafoglio
Logout
Il tuo portafoglio
Saldo: 0.19 wincoin
Transazioni
+0.04
7 gen 2022, alle 17:03:02
+0.05
7 gen 2022, alle 17:03:42
+0.04
7 gen 2022, alle 17:04:12
+0.04
7 gen 2022, alle 17:04:32
+0.02
7 gen 2022, alle 17:05:32


Winsome
user1
Logout
Non hai post. Cl
Nuovo post
Test
Post creato utilizzando l'interfaccia da browser! :D
Nuovo post