WINSOME: a reWardINg SOcial Media

Relazione del progetto

Giorgio Dell’Immagine (601843)

gennaio 2022

Indice

1 Introduzione **4**

2 Librerie di supporto **4**
2.1 Libreria HTTP 4
2.2 Router REST 5
2.2.1 Panoramica generale 5
2.2.2 L'annotazione `Route` 5
2.2.3 L'annotazione `Authenticate` 6
2.2.4 L'annotazione `DeserializzaRequestBody` 6
2.2.5 Esempio di utilizzo 7
2.2.6 Errori 7

3 Architettura del server **7**
3.1 Descrizione generale dei thread e delle strutture dati 7
3.2 Gestione dell'I/O 8
3.3 REST Logic 9
3.4 Database 9
3.4.1 La classe `Wrapper` e le lambda di Java 8 10
3.5 Rappresentazione della rete sociale 10
3.6 Gestione dei token di autorizzazione 11
3.7 Calcolo delle ricompense 11
3.8 Tasso di cambio con BTC 12
3.9 Persistenza 12
3.10 Gestione registrazione e notifiche dei follower tramite RMI 12
3.11 File di configurazione 13

4 Architettura del client **13**
4.1 Il tipo `Result` 14

5 Descrizione dell'API REST **14**
5.1 Filosofia generale 14
5.2 Tabella dell'API REST 15

6 Manuale di utilizzo **16**
6.1 Compilazione 16
6.2 Sintassi comandi client 16
6.3 Esempio di una sessione client 17

7 Descrizione sommaria di tutti i packages **18**

8 Considerazioni e possibili estensioni **19**

1 Introduzione

La componente principale dell'applicativo Winsome è stata implementata con una API REST, che quindi si basa sul protocollo HTTP. Normalmente questi tipi di applicazioni vengono implementati grazie a una libreria apposita che gestisce sia il livello HTTP, sia il livello più astratto dell'API, per fornire al programmatore la possibilità di una descrizione a molto alto livello dell'API senza preoccuparsi dei dettagli della comunicazione. Quello che è stato cercato di fare nel progetto è di costruire una versione minimale, ma non per questo poco potente, di queste astrazioni, che hanno permesso uno sviluppo ad alto livello delle funzionalità.

La scelta di usare REST porta con se diversi vantaggi: primo su tutti il fatto che il protocollo è stateless, quindi si semplifica di gran lunga la gestione dei client da parte del server, in quanto ogni richiesta può essere processata in modo completamente indipendente. Inoltre il mapping delle funzionalità offerte dal sistema con i metodi e le risorse REST è risultato molto naturale.

Essendo la connessione stateless, il controllo della autenticazione è stato implementato grazie all'uso di un token, che il server genera quando un client effettua l'operazione di login, e revoca quando effettua l'operazione di logout. L'autenticazione dei messaggi è stata effettuata grazie all'header HTTP `Authorization`.

La serializzazione dei dati nei corpi delle richieste e delle risposte HTTP è stata affidata alla libreria `jackson`, unica componente esterna utilizzata.

2 Librerie di supporto

2.1 Libreria HTTP

La libreria HTTP implementa un sottoinsieme ristretto delle funzionalità del protocollo HTTP, utile al sistema applicativo Winsome. In particolare la classe astratta `HTTPMessage` rappresenta un messaggio HTTP, e le sue sottoclassi `HTTPRequest` e `HTTPResponse` rappresentano rispettivamente una richiesta HTTP e una risposta HTTP.

Per l'implementazione è stato fatto riferimento alla *RFC 7230*, che descrive il protocollo applicativo HTTP 1.1, anche se chiaramente, come detto sopra, è stato implementato un piccolo sottoinsieme del protocollo.

Le funzionalità implementate sono:

- parsing della start line del messaggio, e quindi parsing del metodo e path nelle richieste, e dello status code nelle risposte

- parsing degli header del messaggio in una apposita `Map<String, String>` che mappa le chiavi al contenuto dell'header
- parsing del contenuto del messaggio, se è presente il campo `Content-length` negli header
- formattazione del messaggio per poter essere spedito.

Tutti gli header diversi da `Content-length` vengono sostanzialmente ignorati da questa componente. Gli header possono comunque essere letti grazie al metodo `getHeaders()`. Solo l'encoding `US_ASCII` è supportato. Gli unici metodi HTTP supportati sono `GET`, `POST`, `PUT`, `DELETE`, che sono i metodi tipicamente usati nelle API REST. Sono supportati solo un numero limitato di response codes, la cui lista completa può essere trovata nella classe enumerazione `HTTPStatusCode`.

2.2 Router REST

2.2.1 Panoramica generale

Il router REST è probabilemente una delle parti più interessanti del progetto. Questa componente fornisce l'implementazione di un router, cioè una componente che prende in ingresso una richiesta HTTP e decide a quale metodo «instradarla» per ottenere una risposta. Anche questa componente implementa un sottoinsieme delle funzionalità di un vero e proprio router, ma è comunque abbastanza potente da permettere una buona astrazione.

Per decidere quale metodo invocare data una richiesa HTTP, vengono usate delle annotazioni (simili a quelle di `JAX-RS`) per decidere quale metodo verrà invocato e per fare il binding dei parametri. I metodi annotati devono essere tutti in una classe. Al momento della creazione del router vengono scanditi i metodi di questa classe alla ricerca delle annotazioni; ciò permette di costruire una mappa di instradamento per le richieste HTTP.

Le annotazioni supportate sono tre: `Route`, `Authenticate` e `DeserializeRequestBody`.

2.2.2 L'annotazione `Route`

L'annotazione `Route` prende due parametri: `method` e `path` e specifica per quale metodo e path deve essere invocata la funzione annotata. Il campo `path` specifica una famiglia di percorsi ammissibili, in quanto un qualsiasi valore racchiuso tra parentesi graffe verrà considerato come un parametro della path. Ad esempio il path `"/a/{param1}/b/{param2}"` contiene due parameri: `param1` e `param2`.

Da uno specifico percorso che viene riconosciuto come appartenente alla famiglia definita, vengono individuati a estratti i parametri attuali, che saranno passati al metodo invocato come argomenti.

I parametri nei percorsi sono «tipati» in base al tipo del corrispondente parametro formale nella *signature* del metodo annotato.

Questo meccanismo del binding è ben spiegato spiegato da un esempio

```
1 @Route(method = HTTPMethod.GET, path = "/a/{param1}/b/{param2}")
2 public HTTPResponse foo(int p1, String p2) {...}
```

Il parametro della path `param1` viene legato all'argomento `p1`, e quindi farà match solo con interi; il parametro della path `param2` viene legato all'argomento `p2`, e quindi farà match con qualsiasi stringa. Nell'esempio sopra la richiesta `GET /a/123/b/foo` scatenerà l'esecuzione di `foo(123, "foo")`, mentre la righiesta `GET /a/foo/b/bar` non verrà accettata, in quanto `foo` non è un intero.

Internamente questo viene implementato trasformando il template in una regular expression, tenendo conto del tipo del parametro.

Gli unici tipi implementati sono **`int`** e `String`, ma il modulo può essere facilmente esteso per supportare altri tipi.

Purtroppo il meccanismo della reflection non permette di leggere il nome dei parametri formali dei metodi a runtime, quindi il nome dei parametri viene praticamente ignorato, l'unica cosa che conta è l'ordine dei parametri e il loro tipo.

2.2.3 L'annotazione `Authenticate`

L'annotazione `Authenticate` indica al router che la richiesta deve essere autenticata.

L'implementazione della funzionalità di autenticazione non viene fornita direttamente dal router, ma viene specificata dall'interfaccia `AuthenticationInterface`, di cui una sua implementazione deve essere fornita al momento della creazione di un oggetto router. Questo è un esempio del principio della *dependency inversion*, tipico della OOP, dove due componenti software (in questo caso il router e la componente che implementa l'autenticazione) non dipendono l'uno dall'altro, ma entrambi dipendono da un interfaccia comune. Questo favorisce il *disaccoppiamento* tra il router e le altre componenti.

Se l'annotazione `Authenticate` è presente, il metodo viene invocato solo e soltanto nel caso in cui l'autenticazione ha successo. L'autenticazione è implementata grazie all'header `Authorization` della richiesta HTTP, come specificato in seguito.

Nel caso in cui la autenticazione abbia successo, l'username dell'utente autenticato viene passato come primo argomento al metodo annotato.

2.2.4 L'annotazione `DeserializaRequestBody`

L'annotazione `DeserializeRequestBody` indica al router che il corpo della richiesta deve essere deserializzato. L'annotazione ha un solo argomento di tipo `Class<? extends RequestModel>`, che

rappresenta il tipo finale dell'oggetto deserializzato, e quindi implicitamente fornisce un controllo di validità del corpo della richiesta, in quanto se la deserializzazione fallisce verrà restituito al client un errore. Da notare che l'argomento deve estendere `RequestModel` in quento tutti i modelli di dati dei corpi delle richieste sono sottoclassi di `RequestModel`. In questo modo possiamo avere un controllo al livello di type system che la classe fornita sia ammissibile come modello dei dati.

Nel caso l'annotazione sia presente, il router passa al metodo annotato il corpo della richiesta deserializzato come ultimo argomento.

2.2.5 Esempio di utilizzo

Per fare un esempio riassuntivo (tratto dalla classe `RESTLogic`):

```
1 @Route(method = HTTPMethod.POST, path = "/posts/{id}/comments")
2 @DeserializeRequestBody(CommentRequest.class)
3 @Authenticate
4 public HTTPResponse commentPost(String callingUsername, int postId,
      CommentRequest reqBody) {...}
```

Il metodo viene invocato per una richiesta `POST` sulle path del tipo `/posts/{id}/comments` dove `id` può essere qualsiasi intero (che viene passato al metodo nella variabile `postId`). Il metodo richiede l'autenticazione, quindi il router usa la sua interfaccia di autenticazione interna per autenticare l'utente; il metodo viene chiamato solo se l'autenticazione ha successo e il nome utente del chiamante viene passato nell'argomento `callingUsername`. Il corpo della richiesta deserializzato in una `CommentRequest` viene passata al metodo nell'argomento `reqBody`.

2.2.6 Errori

Se il router non trova nessun match tra la richiesta e i metodi annotati, allora restituisce al client `404 Not Found`. Se il router fallisce la deserializzazione del corpo della richiesta, restituisce al client `400 Bad Request`. Se viene sollevata una eccezione generica (ad esempio una `IOException`) durante l'esecuzione della richiesta, restituisce al client `500 Internal Server Error`.

3 Architettura del server

3.1 Descrizione generale dei thread e delle strutture dati

Il server mantiene un thread dedicato alle operazioni di I/O, che sono implementate con NIO e canali non bloccanti. Il server soddisfa le richieste concorrentemente, cioè per ogni richiesta fa eseguire

da una threadpool un task worker che la soddisfa. I thread worker accedono a una struttura dati condivisa, il `Database`, che mantiene tutte le strutture dati della rete sociale, gestisce tutta la logica interna e gestisce i problemi di concorrenza tra i vari worker. Ci sono inoltre due thread di supporto, che implementano il calcolo periodico delle ricompense e la persistenza del database, nelle modalità descritte sotto.

3.2 Gestione dell'I/O

L'input/output del server è completamente gestito con NIO e canali non bloccanti. L'implementazione della gestione dell'I/O si può trovare nella classe `RESTServerManager`. Questa componente si occupa non solo di eseguire il multiplexing tramite una `Selector`, ma anche di avviare i thread worker che andranno a processare la richiesta. L'architettura è composta da un singolo thread che si occupa della gestione dell'I/O tramite il multiplexing e molti thread worker che eseguono la logica interna concorrentemente.

Quando un client si connette viene eseguita la `accept` e il client viene registrato nel selector con operazione di lettura. In questa fase l'attachment della `clientKey` è un oggetto di tipo `RequestBuffer`. Questo oggetto mantiene lo stato della lettura della richiesta e implementa funzionalità di parsing parziale della richiesta HTTP.

Questa classe è necessaria perché non è possibile semplicemente leggere i dati di una richiesta su un byte buffer, in quanto a priori non è noto quanto sarà lunga la richiesta HTTP che dovrà essere ricevuta. In primo luogo perché il protocollo HTTP non specifica una lunghezza massima degli header, e in secondo luogo perché la lunghezza del corpo è variabile ed è contenuta nell'header `Content-length`.

Una volta che la richiesta HTTP è stata letta interamente, il client viene rimosso dal selector e viene fatto eseguire dal `ThreadPoolExecutor` interno un worker che andrà a soddisfare la richiesta. Il thread worker (implementato nella classe `RequestExecutor`) invoca il metodo `callAction` del router del server, che andrà a processare la richiesta e ritornerà con una `HTTPResponse`. A questo punto il worker prende la `HTTPResponse`, la formatta e la «lega» (sotto forma di byte buffer) alla `clientKey`. L'interest set del client viene modificato per abilitare le operazioni di scrittura e il selector del server viene svegliato. Questo è necessario perché cambiamenti all'interest set non hanno effetto fintanto che la `select()` non ritorna, e quindi se non venisse svegliato il thread che gestisce l'I/O non inizierebbe la fase di scrittura. Una volta fatto ciò il worker thread termina.

La fase di scrittura è poco interessante, semplicemente viene scritto il contenuto del `ByteBuffer`, che contiene la risposta HTTP, sulla `SocketChannel` del client. Una volta terminata la scrittura il client viene re inizializzato e l'interest set viene riportato a sola lettura.

Un vantaggio dell'avere una architettura REST è che nessuno stato deve esse mantenuto tra le richieste, e così ogni richiesta può essere processata in completa indipendenza.

3.3 REST Logic

La classe `RESTLogic` implementa tutte le funzionalità principali del server Winsome. Si tratta per la maggior parte di metodi annotati (come descritto sopra) che implementano i vari comandi. L'implementazione nei vari metodi è abbastanza lineare, con l'ausilio delle chiamate ai metodi del database, descritte sotto.

3.4 Database

La classe `Database` è la classe che mantiene tutte le strutture dati della rete sociale e implementa tutte le operazioni su queste strutture dati. Per questo motivo è importante che tutte le operazioni che vengono svolte dal database siano safe da un punto di vista della concorrenza.

Il database mantiene essenzialmente tre mappe, la mappa dei post (che mappa id a oggetti `Post`), la mappa degli utenti (che mappa usernames a oggetti `User`) e la mappa dei token di autenticazione (che mappa ogni username al corrispettivo token, se esiste). Queste mappe sono degli oggetti di tipo `ConcurrentHashMap`, quindi sono di per sé safe da usare in un ambiente multithreaded. Siccome però ci sono dei riferimenti incrociati nelle due mappe (ad esempio gli utenti mantengono la lista degli id dei post che hanno scritto) la consistenza tra le due mappe non è banale. Una soluzione naive poteva essere quella di far eseguire tutte le richieste al database in modo sincronizzato, ma così facendo tutto il vantaggio di avere più thread worker che lavorano in parallelo svanisce. La soluzione adottata parte dall'osservazione che, tranne per l'operazione di rimozione del post e di unfollow, i dati non vengono mai rimossi o modificati dalle mappe, ma vengono solo aggiunti. Quando un thread legge quindi un dato da una mappa può essere sicuro che questo dato non verrà rimosso o modificato da un altro thread. Questa assunzione può essere usata per agire concorrentemente sul database sicuri del fatto che non verranno create inconsistenze.

L'operazione di rimozione di un post e di unfollow sono operazioni che modificano o rimuovono dati dal database, e devono quindi essere eseguite in accesso esclusivo all'intero database. La differenziazione tra operazioni normali e operazioni esclusive viene garantita tramite una `ReadWriteLock` mantenuta dal database.

In generale tutte le operazioni che possono essere eseguite concorrentemente devono iniziare e terminare rispettivamente con le chiamate ai metodi `beginOp()` e `endOp()`, che richiedono e rilasciano la read lock. Tutte le operazioni che richiedono accesso esclusivo all'intero database devono iniziare e terminare rispettivamente con le chiamate ai metod `beginExclusive()` e `endExclusive()`, che

richiedono e rilasciano la write lock. Non solo la rimozione di un post ha bisogno di accesso esclusivo al database, ma anche il salvataggio del database e il calcolo delle ricompense.

L'operazione di logout, che potrebbe porre problemi in quanto rimuove un token dalla mappa dei token, in realtà è safe perché la sincronizzazione è garantita dalla `ConcurrentHsahMap`.

3.4.1 La classe `Wrapper` e le lambda di Java 8

In generale tutte le operazioni del database fanno largo uso dei metodi esposti dalla `ConcurrentHashMap`, in particolare del metodo `compute`, che permette di eseguire una sequenza di operazioni in modo atomico in maniera molto semplice. Viene fatto a questo scopo largo uso delle *lambda expression* di Java 8. Queste, pur essendo molto comode, hanno però delle limitazioni importanti: una su tutte è il fatto che non possono catturare l'ambiente (non sono in questo senso delle vere e proprie closures), e possono solo operare su variabili nell'ambiente esterno che siano **`final`** o `effectively` **`final`**. Questo vuol dire che un codice del tipo

```
1 boolean someFlag = true;
2 map.compute(username, (key, value) -> {
3     if(...) someFlag = false;
4 });
```

non è accettabile perché la variabile `someFlag` viene modificata. Viene fatto quindi largo uso dell'oggetto helper `Wrapper<T>` che è un semplicissimo wrapper di valori parametrico con due metodi: `T getValue()` e **`void`** `setValue(T value)`. Questo permette di trasformare il codice sopra in:

```
1 var someFlag = new Wrapper<Boolean>(true);
2 map.compute(username, (key, value) -> {
3     if(...) someFlag.setValue(false);
4 });
```

che adesso viene accettato come codice valido.

3.5 Rappresentazione della rete sociale

Le classi più importanti che mantengono le strutture dati della rete sociale sono la classe `User` e la classe `Post`.

La classe `User` mantiene tutte le informazioni di un utente iscritto a Winsome. In particolare mantiene le informazioni di base come il nome utente, la password e la lista di tag, isieme ad altre informazioni quali i set dei followers e dei followed, il set dei post di cui l'utente ha fatto il rewin e il set di post che l'utente ha creato. Inoltre mantiene le informazioni riguardandi le ricompense, in particolare il valore

del wallet totale dell'utente e la lista di guadagni parziali sottoforma di una `List<PartialReward>` dove la classe `PartialReward` mantiene il timestamp della transazione e il valore della transazione.

La classe `Post` mantiene tutte le informazioni di un post. In particolare mantiene le informazioni di base come l'autore, il titolo e il contenuto, così come il set di utenti che hanno votato il post, il numero di voti positivi, il numero di voti negativi e una lista di commenti. I commenti sono oggetti di tipo `Comment`, che mantengono l'autore e il contenuto. Inoltre vengono mantenute informazioni utili per il calcolo delle ricompense, quali l'età del post (misurata in quante volte il post è stato esaminato per assegnare le ricompense), gli username dei nuovi voti positivi, il numero dei nuovi voti positivi, il numero dei nuovi voti negativi e una mappa che associa gli usernames a quanti nuovi commenti hanno lasciato («nuovi» vuol dire dopo l'ultimo calcolo delle ricompense).

3.6 Gestione dei token di autorizzazione

Il database mantiene una mappa da username a token di autorizzazione, che rappresenta i token validi. Ad ogni utente è associato al più un token. Quando un utente esegue un'operazione di login, il server controlla che non esista già un token associato a quell'utente (in tal caso restituisce errore) e genera un nuovo token casualmente, che viene poi mandato al client. La generazione del token è implementata grazie alla classe `AuthenticationProvider`. Tutti i token sono codificati in Base64. Anche se estremamente improbabile, ogni nuovo token viene controllato per le collisioni con token già presenti nella mappa.

Le richieste HTTP che richiedono autenticazione, devono contenere l'header HTTP `Authorization` nella forma

```
1 Authorization: Basic user:authToken
```

dove `user` è il nome utente e `authToken` è il token di autorizzazione.

Quando l'utente effettua l'operazione di logout, il token viene revocato, in particolare viene rimosso dalla mappa nel database.

3.7 Calcolo delle ricompense

Il calcolo delle ricompense viene eseguito da un thread specifico, che è implementato nella classe `RewardCalculator`. L'esecuzione del thread si compone essenzialmente di un ciclo infinito, dove il thread aspetta per l'intervallo di tempo tra un calcolo e l'altro, richiede l'accesso esclusivo al database e invoca il metodo `calculateRewards()`. A questo punto rilascia l'accesso esclusivo e manda un messaggio multicast a tutti i client notificandoli che le ricompense sono state calcolate.

3.8 Tasso di cambio con BTC

Il tasso di cambio è ottnuto da una richiesta HTTP al server di `random.org`. La richiesta è effettuata al seguente URL:

```
1 https://www.random.org/decimal-fractions/?num=1&dec=10&col=2&format=
    plain&rnd=new
```

Si fa uso del parametro `format=plain` per ottenere solo un numero come corpo della risposta, senza la formattazione HTML.

3.9 Persistenza

All'avvio, il server prova a leggere un file json (il cui percorso è specficato nel file di configurazione) da cui leggere il contenuto del database. Se non ci riesce (per esempio, perché il file non esiste) il database viene inizializzato vuoto.

Il salvataggio del database è gestito grazie a un thread specifico, che è impementato nella classe `PersistenceManager`. Questo thread periodicamente richiede accesso esclusivo al database e salva tutto il contenuto nello stesso file json. L'intervallo di salvataggio è configurabile nel file di configurazione.

La serializzazione è possibile grazie alle versioni `Serializable` di tutte le classi che compongono le strutture dati del database, insieme alle funzioni `cloneToSerializable()` e `fromSerializable()` che clonano e convertono alle versioni `Serializable` per la serializzazione. Questo processo non è sicuramente il più efficiente, ma è molto semplice e efficace.

I token di autorizzazione non vangono salvati sul file, in quanto sono per loro natura temporanei e facilmente riottenibili dal client con una operazione di login.

3.10 Gestione registrazione e notifiche dei follower tramite RMI

Il server esporta e mappa sul registry due oggetti remoti, ai nomi `"Registration-service"` e `"FollowersCallback-service"` che sono rispettivamente l'oggetto che implementa la funzionalità di registrazione e il callback per le notifiche sui follower. Le interfacce comuni tra server e client si possonno trovare nel package `winsome.common.rmi`.

La registrazione viene eseguita dal client ottenendo l'oggetto remoto dal server e invocando il metodo `registerToWinsome(...)`. Se il nome utente esiste già nel database, il metodo lancia una `UserAlreadyExistsException`.

Il meccanismo delle notifiche per i follower è più complicato: il server esporta un `FollowerCallbackService`, che espone i metodi `registerForCallback(...)` e `unregisterForCallback(...)`, che permettono al client rispettivamente di registrare e deregistrare oggetti di tipo `FollowersCallback`. Espone inoltre un metodo `getFollowers()` che permette al client di ottenere la lista iniziale di followers. Nella implementazione del service, cioè nella classe `FollowersCallbackServiceImpl`, il server mantiene una mappa da username a oggetti di tipo `FollowersCallback`.

Il client quindi quando effettua il login, chiama la `getFollowers()` e inizializza così un insieme locale di followers. Il client esporta poi e registra sul server un oggetto di tipo `FollowersCallback`, che ha solo due metodi: `notifyFollowed(...)` e `notifyUnfollowed(...)`; questi permettono al client di essere notificato per cambiamenti all'insieme dei follower. Quando il server deve notificare una modifica al set dei follower di un utente, il server recupera il corrispettivo oggetto remoto `FollowersCallback` e invoca uno tra `notifyFollowed()` e `notifyUnfollowed()`. Il client a questo punto riceve la notifica e aggiorna il suo insieme locale di follower di conseguenza.

Quando il client esegue l'operazione di logout, l'oggetto callback è deregistrato dal server, e l'insieme locale dei follower viene re-inizializzato.

3.11 File di configurazione

Il file di configurazione del server è memorizzato in formato json e contiene tutti i parametri importanti per il funzionamento dell'applicativo. Il file viene letto all'avvio e i parametri vengono usati per configurare il server.

4 Architettura del client

Il client è considerevolmente più semplice architetturalmente del server.

La classe `winsomeConnection` implementa la connessione lato client con il server Winsome. Si occupa di mantenere le informazioni di login (nome utente e token di accesso) e implementa tutte le funzionalità che sono esportate la sistema Winsome. Tutti i metodi che sorrispondono a un comando ritornano il tipo `Result<String, String>`, il quale è descritto sotto.

La classe `CommandLineInterface` implementa il parsing dei comandi da riga di comando e l'esecuzione dei relativi metodi nella `WinsomeConnection`. Il metod principale di questa classe è il metodo `runInterpreter()` che eseguie un ciclo infinito di parsing, esecuzione del comando e stampa dei risultati.

Il client mantiene l'insieme dei follower localmente nella classe `FollowersCallbackImpl`, che si occupa anche di ricevere le notifiche dal server su modifiche all'insieme stesso.

La classe `PresentationUtils` implementa dei metodi che si occupano meramente della formattazione dell'output. In particolare ricevono dei dati in sottoclassi di `ResponseModel` e ritornano una stringa con i dati formattati.

Le notifiche multicast dell'avvenuto calcolo delle ricompense vengono lette da un apposito Thread, implementato nella classe `RewardsNotificationListener`, che essenzialmente esegue un ciclo infinito di ascolto sulla socket multicast. Quando riceve una notifica, avvisa l'utente che le ricompense sono state calcolate.

4.1 Il tipo `Result`

Tutti i metodi che implementano una funzionalità dentro `WinsomeConnection` ritornano il tipo `Result<String, String>`. Questo tipo di dato incapsula un risultato, che può essere un successo o un errore. Questo tipo è vagamente ispirato al tipo `std::result` che troviamo nella standard library di Rust. Il tipo `Result<V, E>` può essere creato in due maniere: con il metodo `ok(V value)`, oppure con il metodo `err(E error)`. Questi corrispondono alla creazione di un risultato di successo e di un risultato di errore. Si può controllare se il valore è un successo o un errore grazie ai metodi `isErr()` e `isOk()`. L'implementazione può essere trovata nella classe `Result` appartenente al package `winsome.lib.utils`

In particolare nel progetto il costruttore `ok` viene usato per ritornare risposte che sono avvenute con successo, mentre il costruttore `err` viene usato per indicare un errore.

Questo tipo di approccio è stato preferito in questa parte rispetto all'approccio «classico» oject oriented, cioè avere delle eccezioni che rappresentano il fallimento, principalmente per semplicità, ma anche per sperimentazione di una metodologia diversa per la gestione degli errori.

5 Descrizione dell'API REST

5.1 Filosofia generale

In generale nel design dell'API è stata adottata estensivamente la filosofia REST. In particolare i metodi `GET`, `PUT`, `POST` e `DELETE` corrispondono alle operazioni semantiche di lettura, modifica, creazione e rimozione. Tutti i path delle richieste sono risorse e mai azioni. L'unica eccezione è la risorsa `/login`, che è vista con il metodo `POST` come la generazione di un nuovo token, mentre con il metodo `DELETE` come la rimozione del token dal database. Queste due operazioni corrispondono alla funzionalità di login e logout.

Le informazioni per il gruppo multicast sono mappate in modo naturale sulla risorsa `/multicast`, che il client può leggere invocandovi il metodo `GET`.

5.2 Tabella dell'API REST

Tabella 1: Tabella delle richieste che non richiedono autenticazione

Metodo e percorso	Funzionalità
`GET /multicast`	ottiene informazioni del gruppo multicast per le notifiche delle ricompense
`POST /login`	effettua il login dell'utente

Tabella 2: Tabella delle richieste che richiedono autenticazione

Metodo e percorso	Funzionalità
`DELETE /login`	effettua il logout dell'utente
`GET /users`	ottiene la lista degli utenti che hanno almeno un tag in comune con l'utente chiamante
`PUT /followers/{user}`	segui l'utente user
`DELETE /followers/{user}`	smetti di seguire l'utente user
`GET /following`	ottieni la lista di persone seguite
`POST /posts`	crea un nuovo post
`DELETE /posts/{idPost}`	rimuovi il post con id idPost
`GET /posts/{idPost}`	visualizza il post con id idPost
`POST /posts/{idPost}/rewins`	fai il rewin del post con id idPost
`POST /posts/{idPost}/rates`	aggiungi un voto al post con id idPost
`POST /posts/{idPost}/comments`	aggiungi un commento al post con id idPost
`GET /posts`	ottieni il blog dell'utente
`GET /feed`	ottieni il feed dell'utente
`GET /wallet`	ottieni il valore del wallet e la lista di transazioni
`GET /wallet/btc`	ottieni il valore del wallet e la lista di transazioni convertiti in BTC

6 Manuale di utilizzo

6.1 Compilazione

Nel progetto sono forniti sue script `bash` (`executeServer.sh` e `executeClient.sh`) che eseguono la compilazione (tramite `javac`) e l'esecuzione rispettivamente del server e del client. I file `.jar` della libreria *Jackson* sono forniti nella cartella `jackson`.

6.2 Sintassi comandi client

- `register <username> <password> [lista di tag]`: esegui la registrazione; la lista di tag è una lista di token separati da uno spazio
- `login <username> <password>`: esegui il login
- `logout`: esegui il logout
- `follow <username>`: inizia a seguire l'utente `<username>`
- `unfollow <username>`: smetti di seguire l'utente `<username>`
- `list users`: ottieni la lista degli utenti che hanno almeno un tag in comune
- `list following`: ottieni la lista degli utenti seguiti
- `list followers`: ottieni la lista degli utenti che ti seguono
- `post <titolo> <contenuto>`: crea un post; titolo e contenuto sono stringhe racchiuse da virgolette
- `rewin <postId>`: effettua il rewin di un post
- `rate <postId> [+1|-1]`: aggiungi un voto a un post; il voto deve essere necessariamente `"+1"` o `"-1"`
- `comment <postId> <contenuto>`: aggiungi un commento a un post; il contenuto è una stringa raggiusa da virgolette
- `blog`: visualizza il tuo blog
- `feed`: visualizza il tuo feed
- `delete <postId>`: rimuovi un post
- `wallet`: visualizza il tuo wallet
- `wallet btc`: visualizza il tuo wallet convertito in BTC
- `exit`: termina il client

6.3 Esempio di una sessione client

Nell'esempio, tutte le linee di input da parte dell'utente sono prefissate da un ">", mentre le linee di output sono prefissate da "<".

```
> register mary mypassword art music sports
< user registered

> login mary mypassword
< ok, auth:xD6Ngu1JSh-njIrQbxk_ywqcczFZkXk_

> blog
< Id    | Author  | Title
< ------------------------------------

> post "my first post" "very good content"
< post created, id:1

> blog
< Id    | Author  | Title
< ------------------------------------
< 1     | mary    | my first post

> logout
< logged out

> register tom securepass art games chess
< user registered

> list users
< Error: UNAUTHORIZED

> login tom securepass
< ok, auth:vMAHVy3IPcb-rv78kMFIvUKDlWzSWAbN

> list users
< User | Tags
< --------------
< mary | art music sports

> follow mary
< user mary followed

> feed

< Id    | Author  | Titolo
< ------------------------------------
< 1     | mary    | my first post


```

```
> show post 1
< Author: mary, postId: 1
< Title: my first post
< Content: very good content
< Votes: positives 0, negatives 0
< Comments:

> rate 1 +1
< post rated

> comment 1 "very good post"
< post commented

> show post 1
< Author: mary, postId: 1
< Title: my first post
< Content: very good content
< Votes: positives 1, negatives 0
< Comments:
<   tom: very good post

> exit
```

7 Descrizione sommaria di tutti i packages

- `winsome.client`: contiene tutte le classi che implemetano le funzionalità del client; contiene la classe `ClientMain`
- `winsome.common`: contiene le interfacce per la comunicazione client-server
- `winsome.common.requests`: contiene i modelli serializzabili dei corpi delle richieste
- `winsome.common.responses`: contiene i modelli serializzabili dei corpi delle risposte
- `winsome.common.rmi`: contiene le interfacce per gli oggetti remoti
- `winsome.lib`: contiene le librerie sviluppate
- `winsome.lib.http`: contiene la libreria HTTP
- `winsome.lib.router`: contiene la libreria che implementa il router REST
- `winsome.lib.utils`: contiene dei tipi di utility generale
- `winsome.server`: contiene tutte le classi che implementano le funzionalità del server; contiene la classe `ServerMain`
- `winsome.server.database`: contiene tutte le classi delle strutture dati contenute nel database, e l'implementazione del database stesso
- `winsome.server.database.exceptions`: contiene tutte le definizioni delle exxezioni lanciate dal database

- `winsome.server.database.serializables`: contiene le versioni serializzabili delle strutture dati del database

8 Considerazioni e possibili estensioni

In generale questo progetto è stato molto interessante, per varie ragioni. Una delle più importanti è che si riesce ad apprezzare come avere un protocollo stateless semplifica molto da un punto di vista architetturale il server, senza però rinunciare alle funzionalità. In particolare ho trovato che questo tipo di architetura funziona molto bene con la gestione dell'I/O con NIO e canali non bloccanti. Inoltre favorisce la concorrenza, in quanto ogni richiesta è completamente indipendente dalle altre.

Una caratteristica particolarmente interessante di questo progetto è stata la sperimentazione con l'utilizzo del meccanismo della reflection di Java, grazie all'implementazione della libreria di routing. Purtroppo l'implementazione finale non è risultata delle più pulite e eleganti, il che è probabilemte dovuto alla mia mancanza di esperienza in questo ambito.

Il progetto è stato sviluppato con la costante idea dell'astrazione e dell'estendibilità: risulterebbe infatti molto semplice aggiungere funzionalità, sia da un punto di vista di API design (il REST in quache modo ci guida molto in questo senso) sia nell'implementazione della funzionalità stessa, che, grazie alle astrazioni implementate, può essere scritta senza troppa fatica.

Un'altra funzionalità che potrebbe essere implementata è un meccanismo di *time to live* dei token, cioè l'invalidazione automatica dei token di autorizzazione se non sono usati per un certo lasso di tempo.

Chiaramente una possibilità è l'estensione delle librerie HTTP e di routing, atte a renderle più complete e aderenti agli standard, ma in questo caso probabilmente sarebbe meglio affidarsi a librerie standard, sicuramente più mature e complete.