Progetto di laboratorio di reti

Samuele Bonini

Indice

1 Architettura di massima del progetto

1.1 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 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 definiti 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 stasso 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.

(img)

Questo documento è strutturato in maniera tale da illustrare, come prima cosa, le caratteristiche delle entità di dominio e quali sono 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.

1.2 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 rispettivamente incarnate 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 un header (`authorization`) contenente un token di 128 bit. La classe `AuthenticationMiddleware` (spiegata più nel dettaglio in 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).

2 Le entità di dominio

2.1 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.

2.2 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 referenziale a livello di 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 dai limiti delle specifiche), 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).

2.3 Classi richiesta e risposta

2.4 Serializer

3 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.

3.1 Strutture dati

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

3.2 Persistenza dello stato

Al momento dell'istanziazione, la classe `DataStoreService` inizializza un thread demone 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, deserializzandolo e, in caso di impossibilità di completare l'operazione (file inesistnete o malformato), restituisce una nuova istanza dello store.

4 Il service layer

Introduzione

4.1 La classe SocialNetworkService

4.2 La classe RewardIssuer

4.3 La classe WalletConversionService

4.4 I servizi di registrazione utente e notifica

5 L'API e il router

5.1 La classe ApiRoute.

5.2 La classe ApiRouter.

5.3 Gestione di una richiesta HTTP.

(img)

6 Il server

6.1 Gestione delle connessioni.

6.2 La classe ServerConfig.

7 Il client CLI

7.1 Gestione di una richiesta al server.

7.2 Rendering dei dati.

7.3 Gestione dei messaggi.

8 Il client GUI

8.1 Realizzazione.

8.2 Limitazioni.