WINSOME: a reWardINg SOcial Media

Andrea Simone Costa - 597287

Introduzione

Sperimentare. La parola chiave dell'intero progetto è "sperimentare", l'obiettivo principale che ha guidato ogni singola scelta durante lo svolgimento della consegna. Si vorrebbe poter sostenere che il seguire questa filosofia abbia portato a svolgere un buon lavoro, ma in realtà l'unico attributo positivo che l'autore sente di poter accostare con un pizzico di immodestia al prodotto finale è "interessante". La lista delle caratteristiche negative, ahimè, è piuttosto prolissa invece.

Pragmaticamente parlando, l'architettura generale si avvicina a quella di un applicativo web moderno. Il server espone delle - alquanto approssimative - REST API, dialoga utilizzando principalmente il protocollo HTTP, come formato di interscambio per i dati si serve del JSON (JavaScript's SON) e impiega token JWT (JSON Web Token) per l'autenticazione e l'autorizzazione degli utenti. La gestione dei client, a più basso livello, è stata demandata a NIO e alle sue peculiarità basate sul multiplexing dei canali. Salendo di qualche livello di astrazione troviamo JExpress (Java Express, o caffè espresso per gli amici), un "framework" per la costruzione di applicazioni web creato da zero per l'occasione e fortemente ispirato al ben più famoso e meglio riuscito Express.js. La CLI, fornita come prima implementazione di un client come da specifica, è invece piuttosto semplicistica, non particolarmente brillante, specialmente per ragioni di tempo, ma in grado di comunicare perfettamente con il server. È stato progettato e implementato, con alcune limitazioni, anche un altro tipo di client: un applicativo frontend eseguibile su un qualsiasi browser.

Il software è stato scritto, nella sua interezza, adoperando il Java SE Development Kit versione 11. Tra le caratteristiche più recenti del linguaggio utilizzate nella stesura del codice troviamo la dichiarazione di variabili tramite la keyword `var`, piuttosto utile per evitare inutile ridondanza, le lambda e gli stream. In generale è stato preferito un approccio dichiarativo anziché imperativo, con forti influenze da parte della programmazione funzionale che hanno preso forma principalmente favorendo l'Either type, e il suo gemello eterozigote Validation, piuttosto che un utilizzo massiccio delle eccezioni.

Concetti base

Si è ritenuto necessario dedicare un piccolo spazio ad una spiegazione veloce, ma essenziale, di alcuni aspetti del progetto che potrebbero risultare non familiari agli esaminatori.

Either<E, A>

L'Either type è spesso utilizzato per modellare, nella programmazione funzionale, una computazione che produce un risultato avente tipo `A` se termina con successo, ma che potrebbe risultare invece in un errore avente tipo `E` in caso di un qualsivoglia problema. Principio cardine del paradigma funzionale è la composizione: dati una entità avente tipo `Either<E, A>` e una funzione `A -> Either<E, B>`, ovvero una funzione che necessita di un parametro di tipo `A`, in caso positivo restituisce un valore di tipo `B`, ma che

potrebbe fallire durante la sua esecuzione producendo un valore di tipo `E`, è possibile ottenere un valore di tipo `Either<E, B>`.

L'operazione fondamentale che permette tale composizione si chiama `flatMap` ed essa opera come descritto a breve. Se l'entità `Either<E, A>` risulta contenere un valore di tipo `A` allora esso viene estratto e passato come argomento alla funzione `A -> Either<E, B>`. Se invece l'`Either<E, A>` contiene un errore `E`, la funzione `A -> Either<E, B>` non viene chiamata in causa e si procede direttamente ad interpretare l'`Either<E, A>` come un `Either<E, B>`: l'errore viene propagato, nessuna altra computazione può avere luogo.

Vediamo immediatamente un esempio:

```
var erequest = HttpRequest.build(HttpConstants.GET)
        .flatMap(req -> req.setRequestTarget("/test/index.html"))
        .flatMap(req -> req.setHTTPVersion(HttpConstants.HTTPV11))
        .flatMap(req -> req.setHeader("User-Agent",
"Mozilla/4.0(compatible; MSIE5.01; Windows NT)"))
        .flatMap(req -> req.setHeader("Host", "www.tutorialspoint.com"))
        .flatMap(req -> req.setHeader("Accept-Language", "en-us"))
        .flatMap(req -> req.setHeader("Accept-Encoding", "gzip, deflate"))
        .flatMap(req -> req.setHeader("Connection", "Keep-Alive"));
```

Ci troviamo di fronte alla generazione di una richiesta HTTP, dove l'Either type incontra il builder pattern e una simil fluent api. Il metodo statico `build` restituisce un valore di tipo `Either<String, HttpRequest>`, ovvero una entità che potrebbe essere una istanza di `HttpRequest` oppure un errore sottoforma di stringa. Tramite l'operazione di `flatMap` la richiesta HTTP prende progressivamente forma. Ogni singola operazione potrebbe fallire a causa di argomenti invalidi, e in tal caso la chain verrebbe immediatamente interrotta. In caso di successo, invece, le proprietà della richiesta in costruzione saranno modificate. È importante osservare che il parametro `req` non ha tipo `Either<String, HttpRequest>`, bensì `HttpRequest`, poiché viene estratto nel caso in cui l'operazione precedente si è conclusa positivamente.

Un valore di tipo `Either<E, A>` può essere generato tramite due costruttori: il metodo `Either.left` per gli errori, il metodo `Either.right` per i successi:

```
public Either<String, HttpRequest> setHeader(String key, String value) {
  if (key != null && value != null) {
    this.headers.put(key, value);
    return Either.right(this);
  } else {
    return Either.left("HTTP request header's key and value cannot be
null");
  }
}
```

Nel progetto, per semplicità, è stato scelto l'uso del tipo concreto `String` per il caso di errore. Questa scelta si è purtroppo rivelata inadeguata al crescere delle possibili situazioni di errore, poiché rende

piuttosto complicato scegliere dinamicamente una strategia piuttosto che un'altra in fase di gestione dell'errore. Una scelta sicuramente più oculata, in tal senso, sarebbe stata `E = Throwable`.

Per quali motivi dovrebbe essere preferito questo approccio piuttosto che il classico lancio delle eccezioni, con conseguente cattura in altri punti del programma? Innanzitutto notiamo che una signature come `A -> Either<Throwable, B>` ben evidenzia il fatto che la funzione potrebbe fallire, e l'utilizzatore della stessa non può, in linea di principio, ignorare la situazione di errore, pena l'impossibilità di compilare a causa delle rimostranze del type system. Quanto appena affermato ha dei lati in comune con il meccanismo delle checked exception.
In secondo luogo è da apprezzare il maggior controllo sulla gestione del possibile fallimento: esso viene elevato a valore e può essere quindi memorizzato, restituito da una funzione o dato come argomento ad un'altra, raccolto in strutture più complesse, elaborato da pipeline dedicate e così via.
Lanciare una eccezione inoltre "distrugge" lo stack fino al gestore dedicato, non è possibile tornare al punto critico per tentare un approccio diverso. La computazione deve essere quindi ripresa dall'inizio, indipendentemente dal suo costo. L'`Either` invece permette di provvedere immediatamente un valore alternativo con cui proseguire la computazione appena fallita.

La modellazione degli errori non è però l'unico scopo per cui esiste l'`Either`, in quanto esso è l'esempio per eccellenza di **sum type**, ovvero una struttura dati contenente un valore che può assumere diversi tipi fissati, e solo uno dei tipi può essere in uso in un dato momento.

Validation<E, A>

Il tipo `Validation` ha delle similarità con il tipo `Either`, ma non possiede lo stesso potere dal punto di vista della ocmposizione. Il prezzo si paga in favore di una necessità profondamente diversa per quanto riguarda il caso di errore, ovvero la possibilità di eseguire comunque delle computazioni, possibilmente in parallelo, nonostante alcune possano fallire, in modo da poter raccogliere il numero più elevato di errori possibile. Il nome di questo tipo è emblematico, infatti esso è spesso utilizzato per modellare controlli di validità su strutture aventi dipendenze esterne, come ad esempio istanze derivanti dalla deserializzazione del JSON ottenuto in base all'input inserito da un operatore umano in un form.

Per poter combinare tra loro i possibili errori durante l'esecuzione delle computazioni, poiché essi potrebbero avere un qualunque tipo `E`, purché sia il medesimo tra tutti essi, la strategia più comune consiste nella loro raccolta in una qualche sequenza, come ad esempio una lista o un array, posticipando completamente la loro gestione. Se infatti è piuttosto chiaro come due istanze di `String` possano essere combinate, lo è molto meno nel caso di due `RuntimeException`. In generale spetterà quindi al client della validazione la decisione sul da farsi coi possibili errori incontrati. Questa tecnica ha un nome preciso per gli adetti ai lavori, si tratta di sfruttare il monoide libero su `E`.

JWT

Passando dalla teoria alla pratica, vediamo brevemente cosa sono i JSON web token e come possono essere utilizzati al meglio per verificare l'identità di un utente.

La documentazione, raggiungibile seguendo il link https://jwt.io/introduction, introduce i JWT nel seguente modo:

> JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be

> verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Un JWT è codificato utilizzando il formato **Base64url** ed solitamente costruito da tre componenti: un header, un payload e la signature:

```
xxxxx.yyyyy.zzzzz
```

L'header è un oggetto JSON codificato contenente informazioni sul tipo di token e sull'algoritmo di firma utilizzato. Anche il payload è un oggetto JSON codificato, nel quale, in linea di massima, possiamo inserire tutte le informazioni di cui abbiamo bisogno per riconoscere l'utente. Infine la firma, la quale è funzione sia dell'header che del payload codificati.

La modalità di utilizzo più comune e semplice consiste nella generazione di un JWT da parte del server al login dell'utente. Il token conterrà gli estremi dell'utente nel payload, sarà firmato utilizzando un secret noto solo al server e verrà inviato all'utente che dovrà utilizzarlo in tutte le richieste successive.
All'atto della ricezione di una richiesta il server controllerà la presenza e la validità del JWT ricevuto, rifiutando categoricamente di procedere in caso uno dei controlli non avesse successo, oppure andando a identificare precisamente l'attore che l'ha generata in caso positivo, grazie alle informazioni contenute nel payload. Il server potrà quindi decidere in quale modalità svolgere le operazioni legate alla richiesta in essere, a seconda ad esempio del ruolo dell'utente.

Il Dominio

Esaminiamo nel dettaglio le varie entità che costituiscono il dominio dell'applicazione.

Comment

La classe `Comment` rappresenta i commenti che gli utenti possono aggiungere ai vari post del social network. Oltre al testo è presente una timestamp, un riferimento allo username dell'autore del commento stesso e un riferimento all'uuid del post corrispondente.
È presente una `CommentFactory` che permette di creare in sicurezza tali entità, dove è strettamente necessario, eseguendo le opportune validazioni.

Feedback

La classe `Feedback` viene utilizzata ovunque il server ritenga opportuno inserire nella risposta HTTP un body, ovviamente in formato JSON. Un feedback è un wrapper contentente un generico dato:

```
{
  res: String, // già in formato JSON
  ok: Boolean
}
```

Post

La classe `Post` rappresenta i post del social network. Ogni post possiede un uuid univoco, un titolo, un contenuto, una timestamp e un riferimento all'autore (username). Sono inoltre presenti tre liste: una per contenere delle informazioni minimali sui rewin del post, una per i riferimenti ai commenti propri del post e una per i riferimenti alle reazioni degli utenti, siano esse positive o negative. Tali liste necessitano di essere sincronizzate manualmente durante l'accesso; poiché sono oggetti sono state utilizzate le lock proprie degli oggetti Java. Infine troviamo un counter che tiene conto del numero di iterazioni già eseguite sul post durante il calcolo delle ricompense, e un flag booleano utile per impedire rewin impropri su un post che è stato appena eliminato.

Lo scopo principale di tale struttura è quello di permettere a più thread aventi compiti diversi di interagire in modo concorrente con la medesima istanza di `Post`. Un thread può quindi aggiungere un commento, mentre un altro thread aggiunge una reazione e un altro ancora aggiunge un rewin. Le uniche due operazioni che devono avvenire obbligatoriamente in mutua esclusione sono l'eliminazione di un post e il rewin dello stesso, altrimenti alcune possibili successioni di avvenimenti potrebbero portare alla non eliminazione dei rewin di un post.
Per quanto riguarda la lettura di tali strutture condivise, viene sempre effettuata una copia in mutua esclusione della lista oggetto dell'operazione. Questo diminuisce ulteriormente la serializzazione delle attività eseguite dai vari thread, anche se ciò potrebbe inficiare la consistenza della lettura in sé in quanto il dato ottenuto potrebbe non corrispondere allo stato ultimo.
Abbiamo a che fare con quella che viene definita **eventual consistency**, nella quale le operazioni di lettura e scrittura hanno disponibilità massima, non è detto che una lettura ottenga lo stato conseguente all'ultima scrittura, ma sicuramente se non vi sono nuovi aggiornamenti ad un dato alla fine tutti gli accessi ad esso restituiranno l'ultimo valore aggiornato.

Per concludere questa sezione dedicata ai post si evidenzia il fatto che i rewin di un post sono stati considerati come istanze quasi completamente separate dal post originale. Viene effettuata una copia del titolo e del contenuto, ma non delle reazioni né dei commenti, che non sono mai condivisi. Ad ogni modo, l'eliminazione di un post ha come conseguenza l'eliminazione ricorsiva di ogni suo rewin.

Infine, è presente una `PostFactory` per la creazione sicura basata su validazioni, dove necessario, di istanze di tipo `Post`.

AuthorPostUuid

Tale classe è un semplice POJO che rappresenta una coppia autore, uuid di un post. È utilizzata principalmente per tenere traccia dei rewin di un post.

Reaction

La classe `Reaction` rappresenta le reazioni che gli utenti hanno sui post del social network. Oltre al un flag che indica se la reazione è positiva o negativa, è presente anche una timestamp, un riferimento allo username di chi ha lasciato la reazione e un riferimento all'uuid del post corrispondente.
È presente una `ReactionFactory` che permette di creare in sicurezza tali entità, dove è strettamente necessario, eseguendo le opportune validazioni.

User

La classe `User` rappresenta gli utenti del social network. Ogni utente possiede uno username, una password (hashed) per eseguire l'accesso al network, una lista di tag, due set per contenere utenti seguiti e utenti che seguono, i quali necessitano di sincronizzazione manuale per l'accesso, e infine una hashmap concorrente per i post che mantiene l'associazione `uuid -> post`.

I medesimi ragionamenti che hanno influito sulle scelte prese nella classe `Post` sono presenti nella classe `User`. Ad esempio, un thread può rimuovere un follower mentre altri thread eseguono diverse operazioni su diversi post, specialmente grazie alle proprietà del tipo di hashmap scelta. Anche le conseguenze, dal punto di vista della consistenza, sono condivise con quelle della classe `Post`.

Infine, è presente una `UserFactory` per la creazione sicura basata su validazioni, dove necessario, di istanze di tipo `User`.

UserTags

Tale classe è un semplice POJO che rappresenta una coppia utente, tags dell'utente. È utilizzata principalmente per restituire ai client la lista di followers/following con i relativi tag.

Wallet

La classe `Wallet` contiene lo stato dei portafogli di ogni singolo utente. Possiede internamente una hashmap concorrente che associa ad ogni utente il proprio portafoglio, ovvero una lista di transazioni. La hashmap è concorrente poiché più thread potrebbero aver necessità di eseguire azioni sul portafoglio di utenti diversi, azioni che non hanno motivo di essere serializzate.
La situazione che necessita di un occhio di riguardo è l'aggiunta di una nuova transazione, da sincronizzare con l'accesso in lettura alla lista di transazioni dell'utente oggetto di interesse.

Valgono le considerazioni sulla eventual consistence fatte in precedenza per altre classi.

WalletTransaction

La classe è un semplice POJO che rappresenta una coppia guadagno, timestamp, e concorre a formare la history del portafoglio di un utente.\ È presente una `WalletTransactionFactory` per la creazione sicura basata su validazioni, dove necessario, di istanze di tipo `WalletTransaction`.

Protocollo HTTP: richieste e risposte

Il protocollo HTTP è un protocollo a livello applicativo usato principalmente come mezzo di interscambio di informazioni nel web, ed particolarmente adatto per le architetture client/server.

HttpRequest

Questa classe espone tutto il necessario per generare una richiesta HTTP base. Dispone di vari metodi, statici e non, che permettono, tra le altre cose, di costruirne progressivamente una, impostando ad esempio il target, gli header e il body, oppure di effettuare il parsing di una stringa contenente una richiesta HTTP valida. Anche l'operazione inversa è supportata: la serializzazione di una istanza `HttpRequest` produce una richiesta HTTP compliant.

HttpResponse

Questa classe, duale della precedente, espone tutto il necessario per generare una semplice risposta HTTP. Anche essa permette la costruzione progressiva, supporta parsing e serializzazione ed espone vari metodi statici per velocizzare la creazione di determinate istanze.

HttpConstants

La classe `HttpConstants` non è istanziabile e ha come unico scopo quello di centralizzare, sotto un namespace comune, le principali costanti che fanno parte del protocollo HTTP, come ad esempio i metodi, i codici di risposta con le corrispondenti reason e i MIME type più comuni.

Content-Length header

È doveroso dedicare un paragrafo, e quale sezione migliore per farlo, ad un problema semplice da porre ma leggermente complicato da affrontare: come può il server (client) determinare, in fase di ricezione di una richiesta (risposta) HTTP, la dimensione dell'eventuale body? Lo standard HTTP delinea tutta una serie di indicazioni, delle quali solo una, in particolar modo, è stata riadattata e implementata nel progetto:

> HTTP/1.1 requests containing a message-body must include a valid Content-Length header field unless the server is known to be HTTP/1.1 compliant. If a request contains a message-body and a Content-Length is not given, the server should respond with 400 (bad request) if it cannot determine the length of the message, or with 411 (length required) if it wishes to insist on receiving a valid Content-Length.

Il server richiede quindi che tutte le richieste HTTP che non siano GET/DELETE/OPTIONS contengano l'header **Content-Length**, in modo tale da poter conoscere esattamente il numero di byte da leggere dopo la sequenza **CR LF CR LF** segnalante la fine della sezione dedicata. In caso contrario, il server si rifiuta di gestire la richiesta e termina unilateralmente la connessione con il client. Da parte sua, il server imposta sempre tale header in ogni risposta che trasmette al client, in quanto anche esso necessita di sapere la dimensione del body della risposta ricevuta.

JExpress

Ogni server che espone una interfaccia REST necessita di poter definire delle rotte, eventualmente parametriche, associando ad ogni rotta un handler dedicato per la gestione delle richieste HTTP ad essa indirizzate. JExpress raggiunge questo obiettivo permettendo una registrazione agile ed effettiva degli handler da associare alle varie rotte, oltre a consentire la schedulazione di più middleware che agiscono prima dell'handler dedicato, feature essenziale per la gestione dell'autenticazione dell'utente. Non è invece possibile registrare più di un handler per ogni singola rotta.

Vediamo un esempio, nel quale viene impostato un handler per la gestione della rotta parametrica `/users/:user_id` sul metodo POST:

```
jexpress.post("/users/:user_id", (request, params, reply) -> {

      assertTrue(request instanceof HttpRequest);
```

```
      assertTrue(params.containsKey("user_id"));

      var response = HttpResponse.build(HttpConstants.HTTPV11,
HttpConstants.OK_200[0], HttpConstants.OK_200[1])
          .flatMap(req -> req.setHeader("Server", "nginx/0.8.54"))
          .flatMap(req -> req.setHeader("Date", "02 Jan 2012 02:33:17
GMT"))
          .flatMap(req -> req.setHeader("Content-Type", "text/html"))
          .flatMap(req -> req.setHeader("Connection", "Keep-Alive"))
          .flatMap(req -> req
              .setBody("<!DOCTYPE html><html><body><h1>User " +
params.get("user_id") + "</h1></body></html>"));

      reply.accept(response);

    });
```

L'handler prende in ingresso tre parametri:

1. `request`, l'istanza della classe `HttpRequest` che descrive la richiesta HTTP
2. `params`, una istanza di `Map<String, String>` che associa ad ogni parametro della rotta il valore attuale
3. `reply`, una istanza di `Consumer<Either<String, HttpResponse>>` ovvero una callback che l'handler dovrà invocare fornendo o la `HttpResponse` da inoltrare al client o una stringa di errore, che verrà automaticamente trasformata in una 500 - Internal Server Error dal framework.

Un middleware viene registrato utilizzando il metodo `use`:

```
jexpress.use((request, params, reply, next) -> {

  request.context = XYZ.parse(request.getBody());

  next.run();
});
```

I middleware sono globali, ovvero entrano in gioco indipendentemente dal metodo della richiesta HTTP da trattare e prima dell'unico route handler dedicato a ciascuna rotta.
Nell'esempio sopra notiamo alcuni aspetti principali. Innanzitutto ogni middleware deve esplicitamente dare il via libera all'esecuzione del successivo, o del route handler, invocando la callback `next`; questo permette a middleware aventi particolari responsabilità di bloccare la successiva parte dell processo gestione della richiesta, avvalendosi anche della possibilità di rispondere immediatamente al client tramite la callback `reply`. Inoltre, ogni richiesta ha un field generico `context`, il quale può essere utilizzato da un middleware per memorizzare dati, come ad esempio un oggetto deserializzato, rendendolo disponibile ai middleware successivi e all'handler dedicato alla rotta.

Possiamo ritenere l'istanza di `JExpress` thread safe, a patto di tenere separate la fase di configurazione dei middleware e degli handler da quella di utilizzo per la gestione delle richieste tramite il metodo `handle`:

```
Either<String, HttpResponse> eresponse =
jexpress.handle(justAnHTTPRequest);
```

Internamente, infatti, il framework non fa uso di né di strutture concorrenti né di blocchi `synchronized` né di lock di alcun tipo, in modo tale garantire la massima reattività. Durante la fase di gestione delle richieste tali strutture dati vengono sempre utilizzante in sola lettura; è solo nella fase di configurazione che si presenta la necessità di eseguire operazioni di scrittura.

In conclusione di questa sezione è doveroso ringraziare tale Mark McGuill che, ormai quasi 6 anni fa, ha eseguito il porting da JavaScript a Java di una utility fondamentale per l'esistenza di JExpress. La trasformazione da rotta parametrica, sottoforma di stringa, a regexp in grado di eseguire il matching sui target delle richieste HTTP, e di estrarre eventuali parametri, è possibile grazie al suo prezioso contributo alla comunità opensource, liberamente fruibile su GitHub.

Utils

Questa sezione raccoglie varie classi e metodi di utilizzo generale.

Hasher

Tale classe espone un metodo statico, `hash`, che esegue l'hasing di una stringa utilizzando l'algoritmo SHA-512.

JWTUtils

Tale classe espone dei metodi per creare, validare e decodificare i token JWT.

Pair

Una generica coppia di oggetti eterogenea.

Triple

Una generica tripla di oggetti eterogenea.

TriConsumer

Una `FunctionalInterface` rappresentante il tipo delle funzioni con tre parametri in ingresso, eventualmente di tre tipi diversi, che restituiscono `void`.

QuadriConsumer

Una `FunctionalInterface` rappresentante il tipo delle funzioni con quattro parametri in ingresso, eventualmente di quattro tipi diversi, che restituiscono `void`.

ToJSON

Una classe che provvede metodi statici utili per la conversione in JSON di alcuni tipi primitivi.

Wrapper

Un banale wrapper che racchiude un qualsiasi tipo di dato. La sua utilità principale è quella di permettere di bypassare alcune semplici limitazioni del linguaggio.