WINSOME: a reWardINg SOcial Media

Andrea Simone Costa - 597287

METTI UN MENU

Introduzione

Sperimentare. La parola chiave dell'intero progetto è stata "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 scritto 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 dalla 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 usare 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.