Http2Java - Documentazione Tecnica

Pellegrinelli Nico, Pellegrinelli Sean, Panzeri Federico  
Versione del 26/01/2024

Sommario

[Contesto 3](#_Toc157160803)

[Requisiti funzionali 3](#_Toc157160804)

[Errori e Warning 4](#_Toc157160805)

[Tecnologie 4](#_Toc157160806)

[Design 5](#_Toc157160807)

[Linguaggio http 8](#_Toc157160808)

[Controlli Semantici 11](#_Toc157160809)

# Indice delle figure

[Figura 1 - Statechart Diagram 3](#_Toc157160501)

[Figura 2 - Package Diagram 4](#_Toc157160502)

[Figura 3 - Class Diagram 5](#_Toc157160503)

[Figura 4 - Sequence Diagram del processo di traduzione 6](#_Toc157160504)

# Contesto

Http2Java nasce dall’esigenza di un team di sviluppatori di rendere più efficiente ed efficacie il proprio lavoro, agendo in particolar modo sulle mansioni più ripetitive e noiose, che inducono spesso i programmatori in errori di distrazione.

In particolar modo è stato identificato il seguente problema: gli sviluppatori del team devono spesso scrivere codice Java per chiamate http, queste chiamate sono spesso molto simili tra loro e di conseguenza lo è anche il codice che le implementa, ciò portava gli sviluppatori a non porre particolare attenzione al lavoro e introdurre svariati errori nelle richieste. Questo problema era accentuato dal fatto che diversi di questi errori non venivano rilevati dalla libreria Java usata per le richieste http e non impedivano l’invio della richiesta stessa (ad esempio, la creazione di una richiesta con l’header “Usr-Agent” invece che “User-Agent” non segnala alcun tipo di errore).

Queste problematiche hanno portato allo sviluppo del compilatore Http2Java, che, tramite un’interfaccia grafica, permette di scrivere richieste http in un apposito linguaggio e di ottenere la traduzione in codice Java, implementando una serie di controlli sulla struttura e il contenuto della richiesta così da fornire utili messaggi di errore allo sviluppatore.

# Requisiti funzionali

La funzionalità principale di Http2Java consiste nel tradurre richieste http (con metodo GET o POST) scritte in un linguaggio ad-hoc in codice Java. Il linguaggio per le richieste, la cui specifica è riportata nella sezione Linguaggio http, deve rispettare la struttura tipica delle richieste http:

* Request line contenete metodo http, path e versione http;
* Una lista di header come coppie chiave-valore;
* Body della richiesta, opzionale.

Http2Java fornisce una semplice ed intuitiva interfaccia grafica che permette di:

1. Scrivere, in un box dedicato, le richieste http nel linguaggio ad-hoc;
2. Importare richieste http da file di testo (alla pressione del pulsante “Import File”) e modificare a piacimento tali richieste (funzione abilitata alla pressione del pulsante “Toggle File”), per mezzo del medesimo box di testo;
3. Alla pressione del pulsante “Translate”:
   1. Se la richiesta http presente nel box dedicato non presenta errori, tradurre la richiesta in codice Java, che viene mostrato in un altro box di testo;
   2. Se la richiesta http presenta uno o più errori, mostrare la lista degli errori e, eventualmente, la posizione dell’elemento che ha generato tale errore (nessun codice Java viene generato);
   3. Mostrare la lista degli eventuali warning, che non impediscono la generazione del codice.
4. Alla pressione del pulsante “Copy”, copiare nella Clipboard il codice Java generato, se presente.

Il funzionamento di Http2Java è riassunto nel seguente statechart diagram:

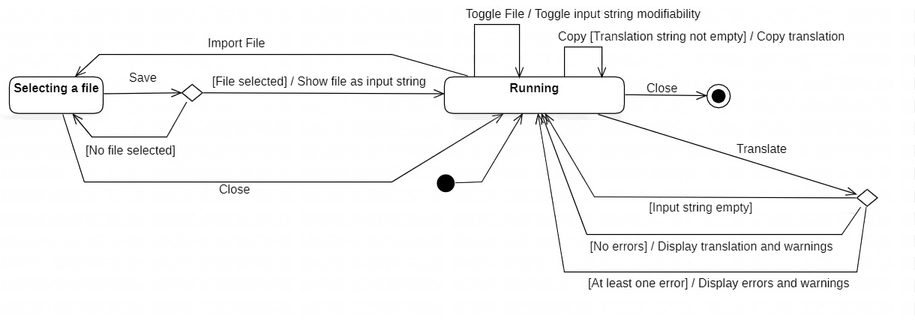


Figura - Statechart Diagram

Http2Java viene distribuito come un eseguibile Jar.

## Errori e Warning

Il focus principale è la rilevazione di errori. In particolare, Http2Java è in grado di riconoscere e segnalare:

* richieste con strutture invalide (che si discostano dalla struttura riportata sopra);
* errori nella request line (sia nella struttura che nel contenuto, come ad esempio path invalido, versione http inesistente o non supportata);
* errori nella definizione di alcuni tra gli header più utilizzati, ovvero “Host”, “User-Agent”, “Content-Type”, “Accept”, “Cookie”, “Authorization”, “Accept-Language”, “Accept-Encoding”, “Cache-Control” e “Max-Forwards”.
* duplicazione di header;
* mancanza dell’header “Host”.

Inoltre, segnala i seguenti warning:

* definizione del body o dell’header “Content-Type” in una richiesta GET;
* mancata definizione del body o dell’header “Content-Type” in una richiesta POST.

# Tecnologie

Http2Java è stato sviluppato con i seguenti tool e tecnologie:

* **Java**: il linguaggio di programmazione dell’intero progetto (l’IDE utilizzato è **Eclipse**).
* **ANTLR 3**: la grammatica per http è stata definita mediante ANTLR 3.4 (con l’aiuto dell’IDE **ANTLRWorks**). Il codice viene generato in Java.
* **Java Swing**: framework orientato allo sviluppo di interfacce grafiche.
* **Apache Maven**: framework per la gestione del progetto Java e la generazione dell’eseguibile in formato Jar.
* **HTTP client API**: libreria, disponibile a partire da Java 11, per l’implementazione di richieste http. Il codice Java generato da Http2Java sfrutta questa libreria.

Il progetto Maven, la relativa documentazione e l’eseguibile si possono trovare nella seguente repository **Github**: <https://github.com/PellegrinelliNico/ProgettoLFC>.

# Design

Il progetto Java è composto da tre package:

* **compiler**: contiene la specifica della grammatica http (salvata nel file http.g) e i file generati a partire dalla stessa (ovvero HttpLexer.java, HttpParser.java e http.tokens). Contiene anche il file SemanticHandler.java contenente la classe che implementa i controlli semantici. In questo package è contenuto quindi il compilatore nella sua interezza (lessico, sintassi e semantica).
* **gui**: contiene i file Gui.java e TextLineNumber.java contenenti le classi che implementano l’interfaccia grafica. La classe Gui contiene il metodo main e implementa l’interfaccia grafica nella sua interezza, fatta eccezione per i numeri di riga presenti nel box testuale http che sono implementati nella classe TextLineNumber.
* **variables**: contiene i file Header.java, RequestLine.java e CompilerError.java, ognuno contenente una classe utilizzata come data type. I data type hanno lo scopo di contenere i dati generati dal compilatore e facilitarne lo scambio e l’utilizzo.

Di seguito viene riportato il package diagram:

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Figura - Package Diagram

Di seguito viene riportato il class diagram:

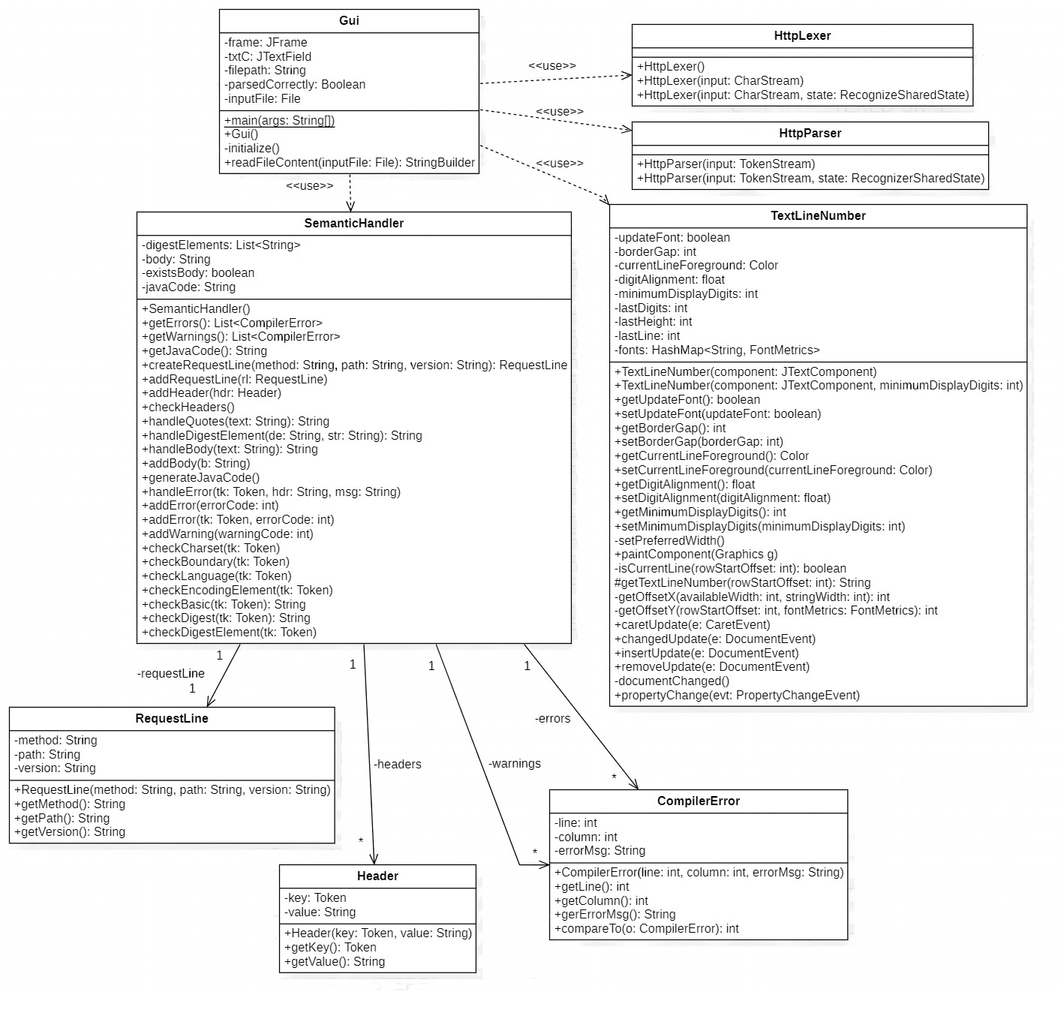


Figura - Class Diagram

Di seguito viene riportato il sequence diagram che illustra la sequenza di operazioni svolte da Http2Java quando viene richiesta la traduzione di una richiesta da http a Java.

Immagine che contiene testo, diagramma, Parallelo, numero

Descrizione generata automaticamente

Figura - Sequence Diagram del processo di traduzione

# Linguaggio http

Di seguito viene riportata la specifica ANTLR che definisce lessico e sintassi della grammatica delle richieste http:

grammar Http;

options {  
 language = Java;  
 k = 1;   
}

request  
 : requestLine  
 header\*  
 body?  
 EOF  
 ;

requestLine  
 : method  
 pathRule  
 HTTP\_VERSION  
 TERMINAL  
 ;

pathRule  
 : PATH QUERY?  
 | STRING  
 ;

method  
 : GET  
 | POST  
 ;

header  
 : hostRule  
 | userAgentRule  
 | contentTypeRule  
 | acceptRule  
 | cookieRule  
 | authorizationRule  
 | acceptLanguageRule  
 | acceptEncodingRule  
 | chacheControlRule  
 | maxForwardsRule  
 | genericHeaderRule  
 ;

hostRule  
 : HOST COLUMN  
 (DNS | IPV4)  
 COLUMN INT\_NUM?  
 TERMINAL  
 ;

userAgentRule  
 : USER\_AGENT COLUMN  
 productRule  
 (productRule  
 (extensionRule)\*)?  
 TERMINAL  
 ;

productRule  
 : PRODUCT  
 PRODUCT\_INFO?  
 ;

extensionRule  
 : PRODUCT  
 ;

acceptRule  
 : ACCEPT COLUMN  
 mimeList  
 TERMINAL  
 ;

mimeList  
 : mimeElement  
 (COMMA mimeElement)\*  
 ;

mimeElement  
 : MIME  
 qValueRule?   
 ;

contentTypeRule  
 : CONTENT\_TYPE COLUMN  
 (MIME charsetRule?  
 |MULTIPART\_MIME boundaryRule)  
 TERMINAL  
 ;

charsetRule  
 : SEMI\_COLUMN ID EQUALS STRING  
 ;

boundaryRule  
 : SEMI\_COLUMN ID EQUALS STRING  
 ;

cookieRule  
 : COOKIE COLUMN  
 cookieList  
 TERMINAL  
 ;

cookieList  
 : cookieElement  
 (SEMI\_COLUMN cookieElement)\*  
 ;

cookieElement  
 : STRING EQUALS STRING  
 ;

qValueRule  
 : SEMI\_COLUMN Q EQUALS Q\_VAL  
 ;

authorizationRule  
 : AUTHORIZATION COLUMN  
 ID  
 (basicAuthRule  
 | digestAuthRule)  
 TERMINAL  
 ;

basicAuthRule  
 : STRING  
 ;

digestAuthRule  
 : authRule  
 (COMMA authRule)\*  
 ;

authRule  
 : ID  
 EQUALS STRING  
 ;

acceptLanguageRule  
 : ACCEPT\_LANGUAGE COLUMN  
 languageList  
 TERMINAL  
 ;

languageList  
 : languageElement  
 (COMMA languageElement)\*   
 ;

languageElement  
 : (ID  
 |LANGUAGE\_ELEMENT  
 |STAR)  
 qValueRule?  
 ;

acceptEncodingRule  
 : ACCEPT\_ENCODING COLUMN  
 encodingList  
 TERMINAL  
 ;

encodingList  
 : encodingElement  
 (COMMA encodingElement)\*   
 ;

encodingElement  
 : (ID  
 |STAR)   
 qValueRule?  
 ;

chacheControlRule  
 : CACHE\_CONTROL COLUMN  
 STRING  
 (COMMA STRING)\*  
 TERMINAL  
 ;

maxForwardsRule  
 : MAX\_FORWARDS COLUMN  
 INT\_NUM   
 TERMINAL  
 ;

genericHeaderRule  
 : STRING COLUMN STRING  
 TERMINAL  
 ;

body  
 : BODY\_STRING  
 TERMINAL  
 ;

Nota: sui token di tipo STRING e BODY\_STRING non viene eseguito alcun tipo di controllo semantico, quindi, ad esempio, se si definisce un header tramite la “genericHeaderRule” non verranno segnalati eventuali errori sul nome o il valore dell’header stesso.

La grammatica decorata (ovvero contenente anche i controlli semantici) può essere consultata al seguente link: <https://github.com/PellegrinelliNico/ProgettoLFC/blob/main/Documentazione/HttpDecorata.g>.

# Controlli Semantici

I controlli semantici implementati, per quanto piuttosto numerosi, sono molto semplici. Tutti riguardano gli header e sono di tre tipi:

1. Verificare che un token sia ammesso come valore di un determinato header, ad esempio nell’header “Accept-Encoding” sono ammessi i valori “gzip”, “compress” e pochi altri.
2. Verificare che i token che definiscono le lingue in “Accept-Language” non abbiano più di 3 caratteri.
3. Verificare che un header non sia definito più di una volta. A tal scopo si mantiene una lista dei nomi degli header già definiti, ogni qual volta il parser riconosce correttamente una headerRule, viene controllato se tale header è già presente nella lista; in caso negativo, il nome del nuovo header viene salvato nella lista, in caso positivo viene generato un errore.
4. Verificare che un elemento dell’header “Authorization” di tipo digest non sia definito più di una volta. Il procedimento è analogo al caso appena descritto.
5. Verificare che l’header “Host” sia definito (questo tipo errore non è associato ad alcun token, quindi nel relativo messaggio non vengono specificate le coordinate dell’errore).

Infine, per quanto riguarda l’ottenimento degli warning si procede come segue: durante il parsing viene tenuto traccia del metodo inserito nella request line (GET o POST) e se l’header “Content-Type” ed il body sono presenti oppure no. Quando queste informazioni sono disponibili si verifica se ci si trova o meno in una situazione da segnalare (come descritto nel capitolo Requisiti funzionali). Anche per gli warning non sono generati da un singolo token, quindi non ne vengono indicate le coordinate.