LPaaS Client API

Marco Galassi
marco.galassi6@studio.unibo.it
Davide Giacomini
davide.giacomini2@studio.unibo.it
Giacomo Venturini
giacomo.venturini2@studio.unibo.it

January 2019

Abstract

Il progetto, sviluppato nell'ambito del corso di Sistemi Distributi, ha lo scopo di facilitare l'utilizzo dell'API REST di LPaaS (Logic Programming as a Service). A tal fine ci è stato richiesto di realizzare (in tre linguaggi) una libreria client che incapsuli la logica delle chiamate HTTPS. In questo modo si può integrare agilmente il service di LPaaS all'interno di future applicazioni.

1 Obiettivo

L'obiettivo del progetto consiste nella realizzazione di un set di librerie client per facilitare l'interazione con l'API REST messa a disposizione da LPaaS (Logic Programming as a Service). Al fine di rendere LPaaS un servizio il più fruibile possibile si è deciso di realizzarne tre versioni in altrettanti linguaggi di programmazione: Java, JavaScript e Python.
Alla realizzazione della libreria si aggiunge il lavoro svolto per perfezionare l'API REST di LPaaS, principalmente dedicato a bugfixing e alla normalizzazione dei dati restituiti, in un formato facilmente manipolabile, quale JSON.

1.1 Scenari

Sebbene l'API LPaaS sia pubblica e qualunque sviluppatore possa quindi scrivere liberamente software per interrogare il servizio tramite chiamate HTTP, riteniamo che la libreria possa rendere molto più agevole il lavoro di chiunque ritenga utile servirsi di

LPaaS nei tre linguaggi citati in precedenza. Il nostro utente di riferimento è pertanto lo sviluppatore che vorrebbe utilizzare l'API LPaaS nella maniera più semplice possibile, ovvero:

- delegando alla libreria la gestione delle richieste di rete
- lasciando alla libreria la gestione di errori
- demandando alla libreria la gestione del token di autorizzazione necessario per alcune richieste
- fornendo il risultato delle chiamate sotto forma di modelli strutturati e immediatamente utilizzabili

2 Analisi dei requisiti

2.1 Ipotesi implicite

Il funzionamento del software prodotto è strettamente legato alla correttezza dell'API REST di LPaaS, in quanto da essa dipendono le strutture dati e i modelli utilizzati per la rappresentazioni di teorie, soluzioni e funtori.

2.2 Requisiti principali

Durante l'analisi sono stati individuati i seguenti requisiti imprescindibili:

- il software deve fornire allo sviluppatore la possibilità di interrogare l'API REST di LPaaS nella sua interezza
- il software deve essere composto da tre versioni scritte in tre linguaggi diversi, al fine di estendere le possibilità di utilizzo
- il software deve essere resiliente in caso di errori e gestire pertanto il possibile fallimento delle richieste HTTP
- il software deve presentare i dati ottenuti in maniera strutturata, rispettando un preciso modello
- il software deve permette allo sviluppatore di configurare le credenziali per accedere alle risorse di LPaaS e gestire la sua autenticazione
- il software, essendo pensato come libreria, dovrebbe essere strutturato e configurato in modo tale che sia agevole importarlo ed utilizzarlo

2.3 Requisiti funzionali

Inoltre, a nostro avviso, una libreria di qualità deve soddisfare i seguenti requisiti:

- il software deve essere semplice da utilizzare e facile da comprendere
- il software deve gestire in maniera completa tutti i possibili casi di errore ed essere sufficientemente esplicativo circa le possibili cause
- il software deve gestire in maniera trasparente modifiche relative alla configurazione del client, senza che lo sviluppatore debba intervenire
- il software deve gestire autonomamente la possibile scadenza e il refresh del token di autorizzazione

2.4 Tecnologie

Sebbene le tre versioni della libreria utilizzino strumenti diversi, tutte basano il proprio funzionamento su un modello che prevede richieste HTTP gestite in maniera asincrona su worker thread e la gestione del risultato tramite callback sul thread principale. Il software fa largo uso di librerie, utili in particolare per la serializzazione/deserializzazione dei dati e per la gestione di base del client HTTP. Non vengono invece utilizzati framework dedicati. Il motivo è principalmente quello di non rendere il nostro software troppo dipendente da altri e mantenere un certo grado di semplicità e flessibilità, evitando pertanto di utilizzare strumenti inutilmente sofisticati per una sola funzionalità richiesta.
Infine per lo sviluppo della libreria JavaScript è stato utilizzato il superset TypeScript. Il principale vantaggio è il supporto alla tipizzazione delle variabili insieme a una migliore gestione del paradigma object oriented. Inoltre, essendo TypeScript compilato in JavaScript, è possibile configurare il compilatore in modo che generi codice compatibile con qualsiasi browser, o conforme ad un particolare standard.

3 Design

3.1 Struttura

Il software si compone idealmente di tre entità:

- Modelli che rappresentano le strutture Prolog, ovvero le risorse esposte dall'API REST (ovvero teoria, goal e soluzione)
- Service (o provider) che hanno la responsabilità di effettuare le richieste all'API REST di LPaaS, divise logicamente in base ai modelli sui quali operano
- Un client, entry point della libreria, che può accedere ai service/provider e richiederne le funzionalità

I service si differenziano in base alle sezioni logicamente distinte definite dall'API REST che gestiscono. In particolare, esisterà un service per ognuna delle seguenti rotte:

- `/theories`
- `/goals`
- `/solutions`

A questi se ne può aggiungere - in base all'implementazione scelta - un ulteriore relativo alla richiesta del token di autorizzazione (`/auth`).
Il client è inoltre modellato (figura 1) in maniera tale da mantenere al proprio interno le informazioni relative all'utente (quali username, password, token, ...) in una struttura dati apposita.

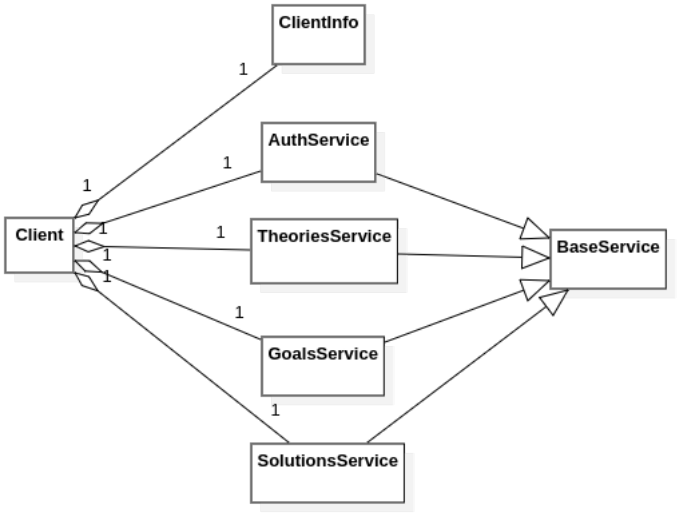

Figura 1: Diagramma UML dell'entry point dell'applicazione e delle entità che lo compongono. La nomenclatura potrebbe variare da implementazione a implementazione, ciò nonostante il modello adottato rimane invariato.

3.2 Comportamento

Il client è l'entry point della libreria, attraverso il quale lo sviluppatore può effettivamente accedere e utilizzare i service. Ha pertanto un duplice compito:

- Mantenere le informazioni dell'utente necessarie per la buona riuscita delle chiamate all'API REST
- Configurare i service dedicati, al fine che siano sempre aggiornati a fronte di modifiche di cui sopra e sempre accessibili

I service raccolgono al proprio interno l'implementazione vera e propria delle richieste HTTP. Forniscono metodi e procedure ben definite per interagire con le risorse contenute sul server e gestiscono il risultato delle chiamate. Due sono i possibili casi:

- La chiamata HTTP ha successo: se è prevista la ricezione di un dato (tipicamente chiamate con metodo GET) questi deve essere convertito in un'istanza di un modello e presentato all'utente tramite callback
- La chiamata HTTP ha fallito: il service deve gestire l'errore, distinguerne il tipo e agire di conseguenza, tipicamente presentando l'istanza di un'eccezione tramite callback dedicata

3.3 Interazione

Client e service interagiscono principalmente in due modi:

- Quando le informazioni delle utente cambiano il client aggiorna o ricrea immediatamente i service in maniera tale che questi le utilizzino come nuovi parametri per le richieste HTTP
- Quando il server comunica tramite errore 401 che il token di autorizzazione è scaduto (in seguito a qualsiasi chiamata che lo richieda) i service interagiscono con il client chiedendo che venga chiamata la rotta per richiederne uno nuovo (`/auth`). Se questa ha successo, il client aggiorna i service e la richiesta di partenza viene ritentata

L'interazione con i modelli avviene in seguito ad una richiesta andata a buon fine che prevede la ricezione di un certo tipo di dato strutturato. Tipicamente tutte le chiamate in GET si comportano in questo modo mentre chiamate in PUT e DELETE sono solite restituire un boolean per determinare il successo dell'operazione di aggiornamento/eliminazione della risorsa.
Ricevuto il dato, la prima cosa da fare è tradurlo in un'istanza (o una lista di istanze) di un modello per poi presentarlo allo sviluppatore nella callback, come spiegato nel punto precedente.

4 Dettagli implementativi

Di seguito per ogni implementazione della libreria elencheremo le rispettive peculiarità implementative.

4.1 Java

Il progetto Java fa uso internamente della libreria Retrofit per la gestione delle richieste HTTPS in maniera asincrona. Il client utilizzato da Retrofit è a sua volta basato dalla libreria OkHttp ed è altamente configurabile. In particolare, viene fatto uso di `Interceptor` per aggiungere automaticamente agli header HTTPS informazioni quali il campo `Accept` (JSON) e il campo `Authorization` (contentente il token di autorizzazione).
Tutti i service sono estensioni della classe astratta `BaseService`, che incapsula la logica necessaria per il setup del client HTTP e mantiene al suo interno riferimenti al `Client`. Ritengo inoltre interessare far notare:

- l'inner abstract generic class `CallbackWrapper`, che permette la gestione centralizzata di ogni richiesta HTTP, compresa la logica necessaria per richiedere l'aggiornamento al client (tramite interfaccia `ClientAuthManager`) in seguito alla ricezione di una risposta con codice 401. L'unico metodo da implementare è `onSuccess()`). In caso di fallimento verrà immediatamente notificata la callback apposita passata al costruttore, già popolata con una eccezione e un messaggio (se presente)

- il metodo `setupService()` tramite cui instauriamo il pattern TemplateMethod.

Ogni service fa riferimento ad una interfaccia (package `routes.interfaces`) che stabilisce metodo HTTP, parametri query e campo body utilizzabili per una certa rotta dell'API REST di LPaaS. Retrofit automaticamente genera un'implementazione di tale interfaccia e il nostro compito si riduce a creare un wrapper per ognuna delle possibili richieste. Ogni metodo wrapper richiede in ingresso una callback, tipicamente di tipo `Consumer<RequestCallback<T>>`, dove `T` è la classe del modello che ci aspettiamo di ricevere dalla richiesta. Per la deserializzazione dei dati è stato utilizzato Gson.
`RequestCallback` è una classe generica che incapsula dentro la duplice funzionalità di fornire un dato (in caso di successo) o un'eccezione (in caso di fallimento).

4.2 TypeScript

Per quanto concerne l'implementazione Typescript, il meccanismo scelto per la gestione asincrona delle richieste di rete è quello delle Promise. Questo perché risulta essere uno tra i meccanismi per la gestione di operazioni asincrone più utilizzati in Javascript e probabilmente quello più elegante e conciso da scrivere.
Permette anche di evitare il fenomeno denominato "callback hell", ovvero la piramide di dichiarazioni di funzioni necessarie per eseguire operazioni asincrone in sequenza. Inoltre, ha notevolmente semplificato l'integrazione con Axios, libreria scelta per eseguire

richieste HTTP, essendo la sua gestione interna dipendente dallo stesso meccanismo. Infine, un altro punto a favore di questa scelta, risulta essere il fatto che combinando le Promise con gli operatori async/await, l'utilizzo di questa libreria risulta estremamente semplificato e conciso, in quanto è possibile utilizzare i metodi asincroni della libreria, come se fossero sincroni.

4.3 Python

Il progetto Python ha come unica dipendenza la libreria *Requests*, che presenta le seguenti caratteristiche:

- rende semplice e agile la gestione delle chiamate HTTP
- è facilmente sostituibile con uno dei suoi superset nel caso in cui si presenti la necessità di utilizzare ulteriori funzionalità

Per l'implementazione delle chiamate alla REST API è stato utilizzato un approccio callback-based, effettuando un wrapping dei metodi messi a disposizione da *Requests*, dato che è quello prediletto nei contesti asincroni.
Infine, per superare l'assenza di meccanismi di overload (limitazione tipica dei linguaggi di scripting) sono stati utilizzati:

- parametri di default – nella definizione dei metodi
- classmethod – per avere un modo per inizializzare un oggetto con parametri diversi rispetto a quelli richiesti dal metodo `__init__`

5 Validazione

Le tre versioni non condividono le stesse tipologie di test. Qualora ritenuto necessario, si è deciso di:

- Creare dei *mock* e di testare la gestione di possibili risultati attesi (successo, fallimento, codici di errore, ...) non potendo fare test automatizzati relativi a vere richieste HTTP (poiché soggette all'indeterminismo della rete)
- Scrivere test relativi alla corretta configurazione del client e al suo aggiornamento in seguito alle sue modifiche (URL base, username, password e token di autorizzazione)

6 Istruzioni e deployment

Di seguito elencheremo i requisiti e gli step necessari per poterle testare in un nuovo environment (anche inclusi all'interno dei rispettivi Readme.md presenti in ciascun progetto).

6.1 Java

Il progetto Java necessita della versione 8 installata sul proprio calcolatore. Fa uso inoltre del build tool Gradle per la gestione delle dipendenze e si consiglia l'utilizzo del wrapper attualmente presente per le fasi di building, testing e running.

6.2 TypeScript

Il progetto typescript è configurato come *package npm*, questo significa che è possibile aggiungerlo come dipendenza ad un qualsiasi progetto javascript/typescript utilizzando il package manager **npm**, tramite il comando `npm install PATH`, dove `PATH` rappresenta la directory nel quale risiede questo progetto. Essendo l'utilizzo di npm molto agevole si è deciso di non utilizzare gradle per questa parte del progetto.

6.3 Python

Per configurare l'ambiente Python è necessario avere installato:

- Python 3 - il modulo è stato sviluppato e testato utilizzando la versione 3.6.7 del linguaggio
- pip 3 - il package manager di Python
- Java 8 o superiore se si vuole utilizzare Gradle

Inizialmente è stato utilizzato il plugin PyGradle, abbandonato in seguito a causa dell'osticità del suo setup con le sue versioni più recenti e il supporto parziale dei sistemi Windows. Si è deciso quindi di passare a una configurazione basata su script/batch files, utilizzando Gradle semplicemente come strumento per lanciarli.
Per evitare l'installazione globale delle dipendenze necessarie a testare il modulo Python è stato utilizzato *venv*, che permette di creare un ambiente virtuale isolato.

- il comando `./gradlew build` esegue lo script *setup-environment* che installa le dipendenze necessarie ed esegue i test
- il comando `./gradlew genDistPackage` esegue lo script *generate-distribution-package* che crea nella directory *dist* il package per la distribuzione del modulo
- la sua installazione (globale) può essere fatta attraverso il comando `pip3 install /path/to/dist/lpaas-api-python-1.0.tar.gz`

7 Esempi di utilizzo

Essendo tre librerie differenti, per ognuna mostreremo un esempio di utilizzo.

7.1 Java

```java
import routes.client.Client;
import routes.client.ClientInfo;

/* Creating the client info with a builder */
ClientInfo clientInfo = new ClientInfo.Builder("https://www.domain.com/lpaas/")
          .setUsername("username")
          .setPassword("password")
          .build();

/* Creating the client */
Client client = new Client(clientInfo);

/*
* Example of a server request:
* retrieving the current solution list through the SolutionsService
*/
client.getSolutionsService().getSolutions((requestCallback) -> {
  if (requestCallback.getResult().isPresent()) {
    // We received a successful response providing a list of Solution
    List<Solution> solutions = requestCallback.getResult().get();
    ...
  } else {
    // Something went wrong, we got nothing
    if (requestCallback.getThrowable().isPresent()) {
      // Checking throwable, an exception may be present
      Throwable throwable = requestCallback.getThrowable().get();
      System.out.println(throwable.getMessage());
    } else {
      // Generic error with no exception
      System.out.println("Something went wrong");
    }
  }
});
```

7.2 TypeScript

```typescript
import { LPaaS } from 'lpaas-js';

//configure the base url
let lpaas = LPaaS.configure({ baseUrl: 'https://www.domain.com/lpaas' });

//execute login
await lpaas.login({ username: 'test', password: 'test'});

//retrieve the goal provider
let goalProvider = lpaas.goals();

//retrieve all goals
let goals = await goalProvider.getGoals();

//add new goal
await goalProvider.addGoal('test', 'test(1)');

//remove a goal
await goalProvider.deleteGoal('test');
```

7.3 Python

```python
from src.main.python.client.Client import Client
from src.main.python.routes.RequestCallback import RequestCallback

# creates library entry point
client = Client("https://www.domain.com/lpaas", "username", "password")

# retrieves the authentication service
auth_service = client.get_authentication_service()
# not really necessary, this is automatically done if a call fails due
 ↪  to expired token
auth_service.refresh_token()

# retrieves the theories service
theories_service = client.get_theories_service()

# set a callback to retrieve result
def callback(rcb: RequestCallback):
    println(rcb)

# retrieves the default theory
theories_service.get_theory('default', callback)
```

8 Conclusioni

Per concludere, si riassumono le macro task svolte durante lo svolgimento del progetto:

- controllo ed eventuale correzione dell'implementazione del servizio LPaaS.
- normalizzazione ed aggiornamento dei dati restituiti dall'API REST di LPaaS.
- sviluppo di una libreria client per poter interagire con il servizio LPaaS in tre versioni differenti (in linguaggio Java, JavaScript e Python), al fine di poter essere facilmente integrata in progetti di più larga scala.

Per vedere lo stato del progetto al momento del suo completamento si faccia riferimento al seguente commit per quanto riguarda LPaaS e al tag v1.0.0 per quanto riguarda la libreria sviluppata.

8.1 Lavoro futuro

Eventuali bug presenti lato server nell'API REST di LPaaS non sono stati sistemati, in quanto tale lavoro esula dal nostro obiettivo principale. In particolare sentiamo il dovere di segnalare come l'API sembri gestire i parametri query o il body di alcune chiamate in maniera non sempre consistente e questo può determinare eccezioni inaspettate. Ad esempio alcune richieste si aspettano nel body testo Prolog correttamente formato e terminante con un "." mentre altre, per lo stesso input, potrebbero sollevare un'eccezione di "sintassi non corretta".