UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

Università di Verona

A.A 2018/2019

Esercitazione REST

Creazione di REST web services tramite Swagger

Sohail Mushtaq

13 giugno 2019

Indice

Capitolo 1

Introduzione

1.1 Architetture orientate ai servizi

L'architettura orientata ai servizi (Service Oriented Architecture o *SOA*) definisce un nuovo modello logico secondo il quale sviluppare il software. Tale modello è realizzato dai **Web Service** (WS), che si presentano come moduli software distribuiti i quali collaborano in maniera standard attraverso un canale web. Il Web Service è una funzionalità messa a disposizione in modalità server ad altri moduli software implementati come client. I componenti software **interagiscono tra loro attraverso protocolli Web Service (ad es. REST, SOAP) trasportati in connessioni HTTP (da cui il nome Web Service)**. E' compito del protocollo Web Service definire una forma serializzata dei parametri attuali da passare al web service e dei valori restituiti da quest'ultimo. Quando uno sviluppatore crea un Web Service, si deve preoccupare di definire:

- la **logica di funzionamento del servizio**, ovvero quello che dovrà fare; questo può spaziare da una semplice classe ad un'applicazione molto complessa;
- il **web container** su cui verrà installato (ad es. Apache, Tomcat, Jboss, Glassfish) e che ne consentirà l'uso da parte dei client.

1.1.1 Vantaggi tecnologici

La nuova struttura collaborativa distribuita porta ad una serie di vantaggi tecnologici che riassumiamo brevemente.

- **Software come servizio**: Al contrario del software tradizionale, una collezione di metodi esposta tramite Web Service può essere utilizzata come un servizio accessibile da qualsiasi client.

- **Interoperabilità**: I Web Service consentono l'incapsulamento; i componenti possono essere isolati in modo tale che solo lo strato relativo al servizio vero e proprio sia esposto all'esterno. Ciò comporta due vantaggi fondamentali: indipendenza dall'implementazione e sicurezza del sistema interno. la logica applicativa incapsulata all'interno dei Web Service è completamente decentralizzata ed accessibile attraverso Internet da piattaforme, dispositivi, sistemi operativi e linguaggi di programmazione differenti.

- **Semplicità di sviluppo e di rilascio**: Un'applicazione è costituita da moduli indipendenti, interagenti tramite la rete; la modularità semplifica lo sviluppo. Rilasciare un WS significa solo esporlo al Web.

- **Semplicità di installazione**: la comunicazione avviene grazie allo scambio di informazioni in forma testuale all'interno del protocollo HTTP usato per il Web e utilizzabile praticamente su tutte le piattaforme hardware/software. I messaggi testuali viaggiano sullo stesso canale utilizzato per il Web e quindi sono compatibili con firewall e NAT. La comunicazione tra due sistemi non deve essere preceduta da noiose configurazioni ed accordi tra le parti.

- **Standard**: concetti fondamentali che stanno dietro ai Web Service sono regolati da standard approvati dalle più grandi ed importanti società ed enti d'Information Technology al mondo.

1.1.2 Motivazioni

Si sta osservando un sempre maggiore utilizzo dell'approccio SOA nella creazione di applicativi software per i seguenti motivi:

- **Protezione della Proprietà Intelletuale**: il cuore dell'applicazione rimane sul server e non viene distribuito neanche come eseguibile agli utenti.

- **Requisiti di potenza di calcolo**: la potenza di calcolo richiesta per l'applicazione può essere soddisfatta dal server senza gravare sulla CPU e il consumo energetico del client (ad es. smartphone).

- **Requisiti di memoria di massa**: la grande memoria di massa richiesta per l'applicazione può essere soddisfatta dal server senza gravare sulle risorse del client.

- **Comodità di distribuzione agli utenti**: ogni utente usa un software client minimale (eseguibile per PC, plugin nel browser web, app per smartphone) che si connette al server. Questo comporta le seguenti conseguenze:

  - **Aggiornamento istantaneo**: la logica applicativa interna al server può cambiare in qualsiasi momento (ad es. per eliminare bug, adeguamenti normativi di un SW legale, ecc.) senza dover re-distribuire aggiornamenti agli utenti che si trovano istantaneamente ad utilizzare il software aggiornato.
  - **Sviluppo e mantenimento di una sola versione del prodotto** a fronte di molteplici piattaforme utente (ad es. Windows, Linux, MAC, smartphone).
  - **Maggiori ritorni economici** col nuovo approccio pay-per-use rispetto al tradizionale approccio della distribuzione e installazione dell'applicativo sui PC degli utenti. Eliminazione del fenomeno della pirateria.

1.2 REST

REpresentational State Transfer (REST) è uno stile architetturale ("architectural style") che permette ai sistemi computer di comunicare tra di loro in modo semplice. In particolar modo possiamo dire che REST:

- **non è un protocollo** in quanto non descrive i messaggi esatti (o parti di essi) che i sistemi devono scambiarsi tra di loro. Piuttosto REST specifica i requisiti (vincoli architetturali) che le parti del sistema devono soddisfare.

- **non è una specifica** in quanto a differenza dei servizi Web basati su SOAP, non esiste uno standard "ufficiale" per le API Web RESTful. Questo perché REST è uno stile architetturale, mentre SOAP è un protocollo. REST non è uno standard in sé, ma le implementazioni REST fanno uso di standard, come HTTP, URI, JSON e XML.

- **non è per forza legato ad HTTP** in quanto nei vincoli REST non c'è nulla che rende obbligatorio l'uso di HTTP come protocollo di trasferi-

mento. È perfettamente possibile utilizzare altri protocolli di trasferimento come SNMP, SMTP e persino altri non basati sull'architettura TCP/IP.

Un servizio web che utilizza i metodi HTTP e implementa i principi REST viene chiamato **RESTful**.

1.2.1 Principi

Un'architettura REST si basa sui seguenti principi.

Client-Server

Nell'architettura REST chi utilizza il servizio agisce con la modalità del *client* che fa il primo passo richiedendo qualcosa alla controparte *server* che si trova già precedentemente in ascolto e che fornisce una risposta. L'unico requisito è che entrambi gli attori (client e server) sappiano in che formato scambiarsi i messaggi. E' importante precisare che il client è un programma scritto ad-hoc per utilizzare un certo servizio e non uno strumento di uso generale come un web browser. In altre parole, client e server di una architettura REST sono due moduli di un unico programma che comunicano attraverso la rete in un canale web come appunto previsto nel paradigma web service.

Stateless

La comunicazione tra client e server deve essere senza *stato* tra richieste/-risposte successive. Ciò significa che ogni richiesta da parte di un client dovrebbe contenere tutte le informazioni necessarie per il server per comprendere il significato della richiesta e fornire la risposta. Inoltre tutti i dati sullo stato della sessione dovrebbero poi essere restituiti al client nella risposta. In breve ogni richiesta è come se fosse la prima richiesta e non dovrebbere essere correlata ad una precedente richiesta. Se il server non conserva lo stato, è più leggero e può scalare molto più facilmente sul numero di richieste che può servire. Inoltre il server può comprendere una richiesta senza aver visto quelle precedenti. Questo elimina alla radice il problema della gestione e sincronizzazione delle sessioni utente in ambienti clusterizzati. In altre parole, lato server può esistere una *batteria di macchine* ed ogni richiesta può essere istantaneamente girata ad uno qualsiasi dei server presenti secondo una logica di bilanciamento di carico (*load balancing*)

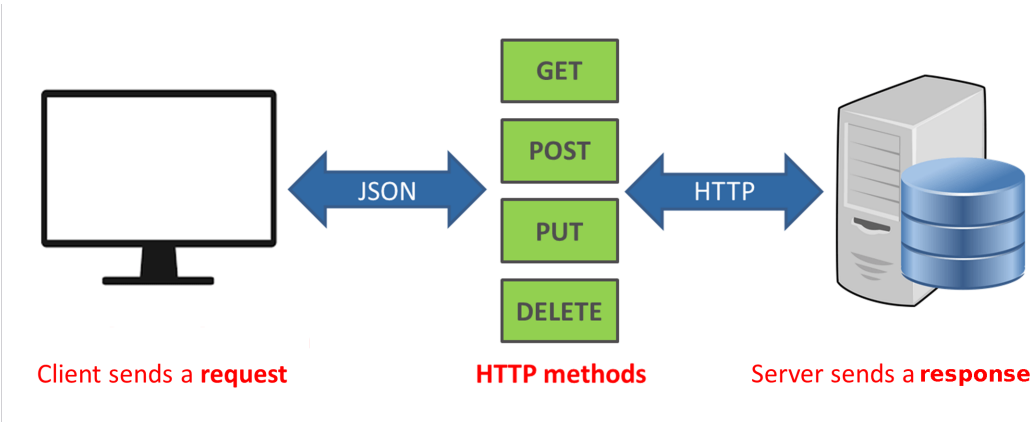

Figura 1.1: Architettura REST

1.2.2 Comunicazione tra Server e Client

Nell'architettura REST il client invia una richiesta per recuperare o modificare una risorsa e il server invia una risposta.

Richiesta

Una richiesta generalmente è composta da:

- **metodo di richiesta** che definisce che tipo di operazione si vuole fare. Ci sono 4 tipi di richieste HTTP che possiamo utilizzare per interagire con le risorse in un sistema REST:
  - **GET**: recupera una specifica risorsa tramite *id* o un'insieme di risorse;
  - **POST**: crea una nuova risorsa;
  - **PUT**: aggiorna una risorsa specifica identificata tramite *id*;
  - **DELETE**: elimina una risorsa specifica identificata tramite *id*.
- un **header** che permette al client di aggiungere informazioni relative alla richiesta. All'interno, inoltre, il client specifica il tipo di contenuto che è in grado di ricevere dal server, ad esempio:
  - **text/html** per ricevere un file testo contenente HTML;
  - **text/plain** per ricevere un file di testo;
  - **application/json** per ricevere la risposta in un JSON.

- il **percorso della risorsa**, sotto forma di *Uniform Resource Identifier (URI)*, a cui è indirizzata la richiesta. Se ad esempio volessimo leggere la lista dei prodotti presenti su un catalogo, verrà semplicemente invocato il metodo HTTP GET sulla URI che rappresenta il catalogo prodotti (si noti che siccome il metodo GET è quello usato per default dai browser web nell'accesso alle pagine, in questo specifico esempio si può simulare il comportamento del client REST aprendo con il browser la pagina `http://www.mionegozio.it/lista/prodotti`). È cura del progettista del server far corrispondere l'URI lista/prodotti all'opportuna query sul database.

- un **message body** contenente dati, in generale opzionale ma necessario nel caso dei metodi POST e PUT con cui si passano dei dati al server.

Risposta

Il server invia l'informazione richiesta nel *body* del messaggio HTTP e riporta le seguenti informazioni nell'header:

- **content-type** che indica al client il tipo di dati inviati nel corpo del messaggio;

- **response code** che indica al client se l'operazione richiesta ha avuto successo o meno. Alcuni dei codici di risposta sono i seguenti:

```
200 (OK)
201 (CREATED)
204 (NO CONTENT)
400 (BAD REQUEST)
403 (FORBIDDEN)
404 (NOT FOUND)
500 (INTERNAL SERVER ERROR)
```

1.3 Stub e Skeleton

Si definisce *API RESTful* un'insieme di chiamate HTTP (metodi HTTP, URI da utilizzare, formati dei dati passati e restituiti) che rappresenta l'interfaccia (API=Application Program Interface) ad un insieme di servizi web di tipo REST. Client e server, oltre ad implementare le funzionalità applicative di propria competenza in riferimento all'applicazione globale, devono anche contenere il codice che implementa chiamate e risposte REST. Il codice lato client viene chiamato *Stub* mentre il codice lato server è chiamato *Skeleton*. Tale codice è dipendente della specifica API ma tale relazione di dipendenza è formalizzabile rigidamente e quindi è possibile generare automaticamente

il codice di stub e skeleton evitando noiose e ripetitive operazioni manuali. Il resto del documento riguarda proprio questo aspetto.

1.4 Swagger

Swagger è un insieme di specifiche e di strumenti software che mirano a semplificare e standardizzare i processi di documentazione di API RESTful. Il cuore di Swagger consiste in un **file testuale** (in formato sia YAML sia JSON) dove le API RESTful che devo utilizzare sono descritte in un formato studiato per essere interpretabile correttamente sia dagli esseri umani sia da uno strumento software. I vantaggi di questa standardizzazione sono:

- una più obiettiva e condivisibile **documentazione delle funzionalità** di un'API;
- la possibilità di **generare codice stub/skeleton** sfruttando direttamente le specifiche definite nello schema.

Infatti, i metadati presenti nel file forniscono informazioni sufficienti per generare sia lo stub sia lo skeleton comprensivo dei percorsi HTTP e la validazione degli input. La generazione automatica dello stub è vantaggiosa perché rende possibile un'adeguamento veloce del codice sorgente del client alle evoluzioni dell'API. La generazione della parte server è vantaggiosa in quanto è possibile concentrarsi direttamente sulla programmazione delle funzionalità centrali del servizio erogato (la cosiddetta *business logic*) senza perdersi nei dettagli ripetitivi della gestione delle chiamate REST.

1.4.1 Swagger framework tools

Swagger fornisce alcuni tool per progettare API e migliorare il lavoro con le web services.

- **Swagger Editor** (figura 1.2) è un editor online che permette di documentare, progettare, modificare API. Inoltre, offre la possibilità di generare il codice del server e del client.

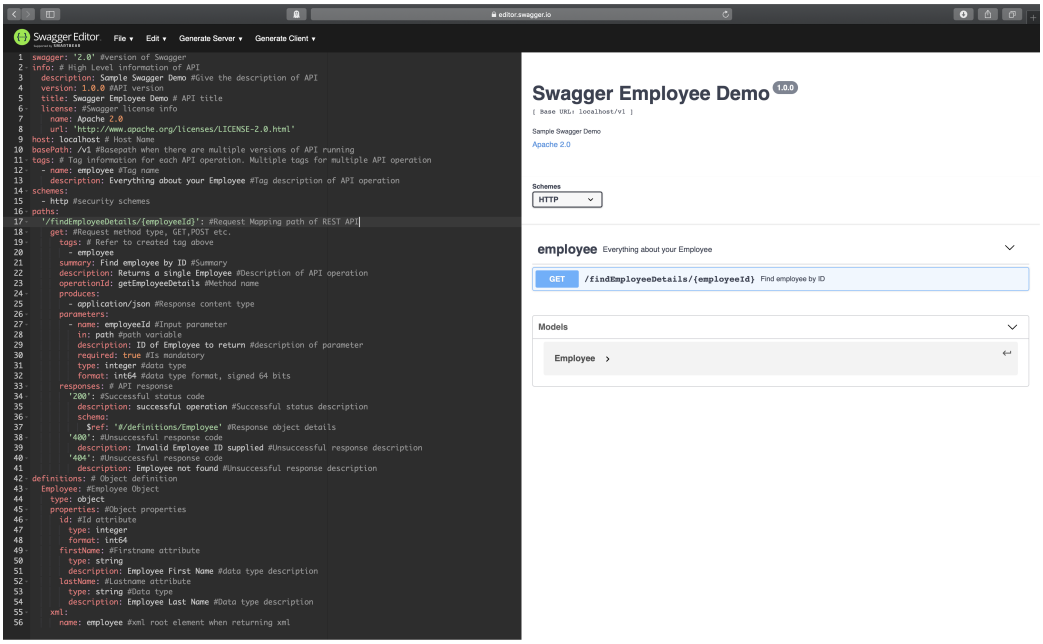

Figura 1.2: Swagger Editor

- **Swagger Codegen** (figura 1.3) permette allo sviluppatore di generare il codice del client per piattaforme diverse. Se non si vuole utilizzare Swagger Editor, strumento online, per generare il codice server e client è possibile farlo anche tramite Swagger Codegen che è uno strumento offline.

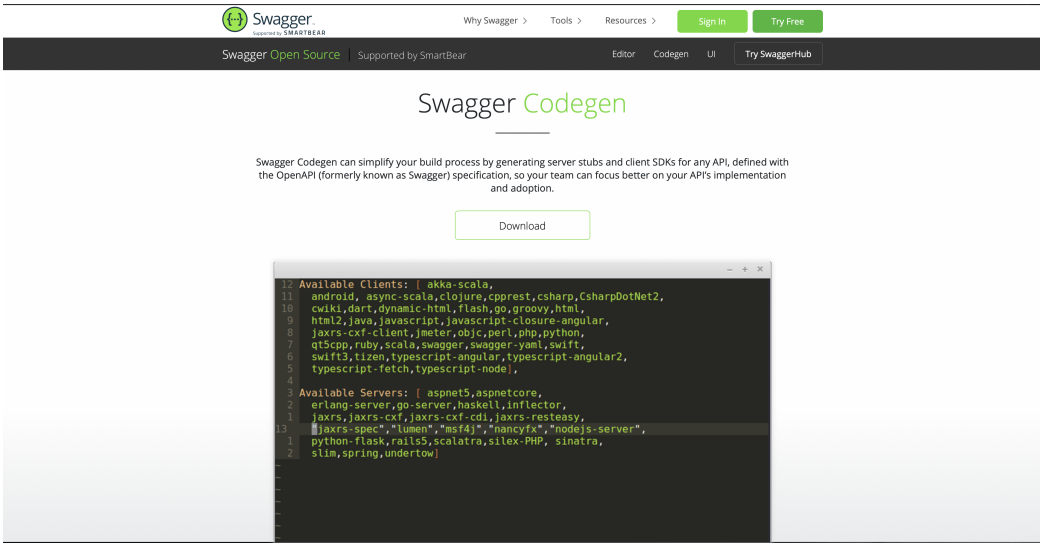

Figura 1.3: Swagger Codegen

- **Swagger UI** (figura 1.4) permette allo sviluppatore di visualizzare le API tramite un'interfaccia web semplice. Inoltre, offre anche la possibilità di provare il funzionamento delle API.

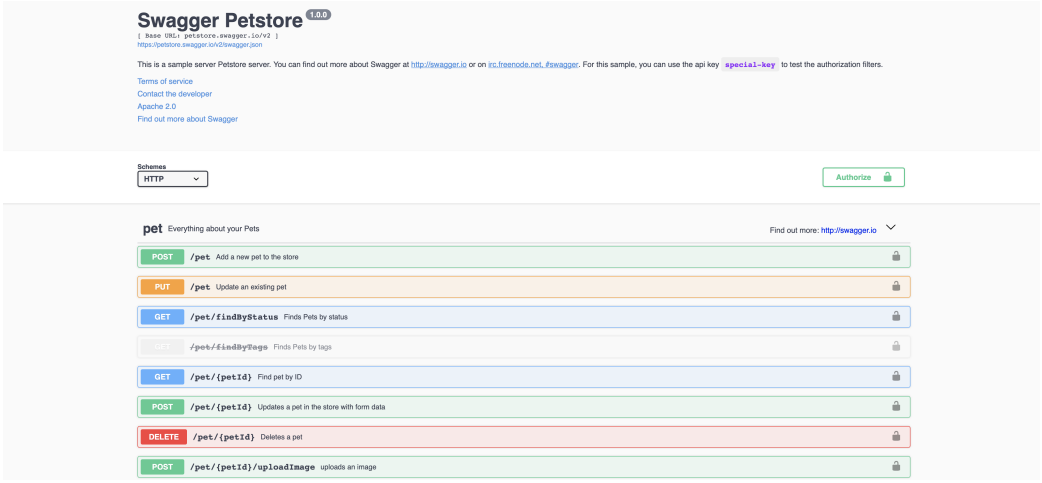

Figura 1.4: Swagger UI

- **Swagger Inspector** (figura 1.5) permette di testare le API generate in modo semplice.

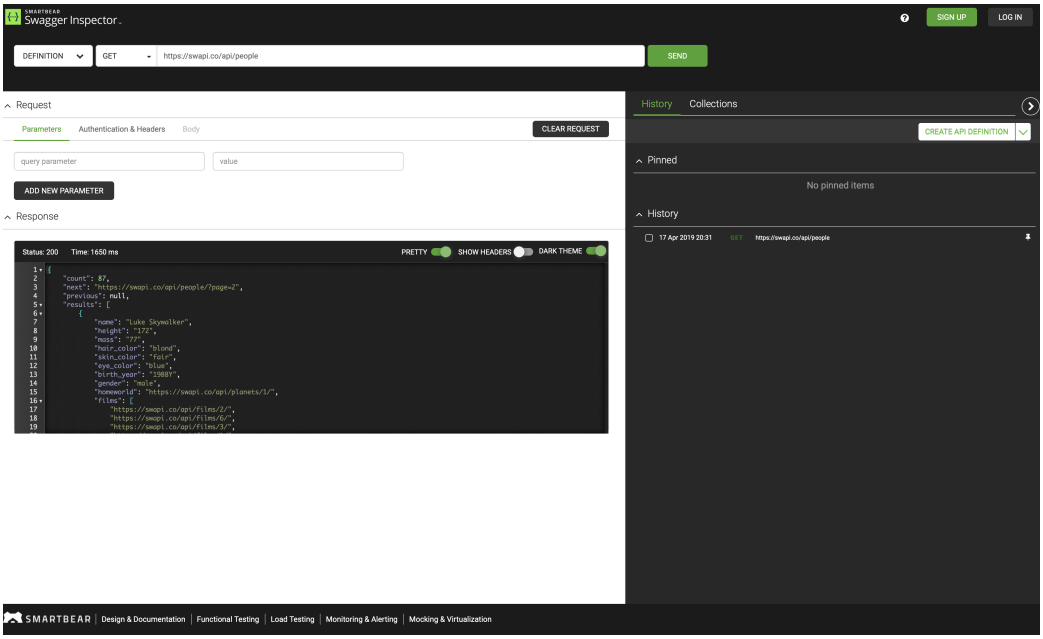

Figura 1.5: Swagger Inspector

1.5 Obiettivo

L'obiettivo di questa esercitazione è quello di fare esperienza del flusso di progettazione delle API REST utilizzando il framework Swagger. La web service che andremo a realizzare è una semplice calcolatrice che accetta le operazioni di somma, sottrazione, moltiplicazione e divisione tra due operandi (figura 1.6). Una volta generato il codice del server e client andremo a "riempirli" opportunamente per svolgere i compiti richiesti.

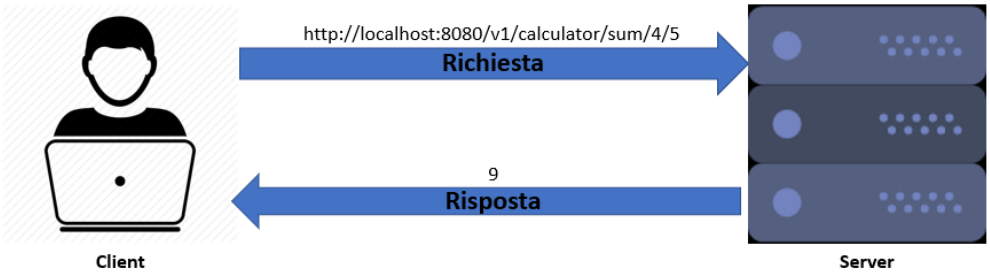

Figura 1.6: Calcolatrice

Capitolo 2

Setup

2.1 Requisiti

Questa esercitazione richiede che sia installato sulla propria macchina lo strumento Eclipse IDE.

2.2 Contenuto esercitazione

Il file zip scaricato contiene il file `calcolatrice.yaml` che descrive l'API che vogliamo realizzare lato server e usare lato client. Inoltre, sono stati aggiunti dei commenti per migliorare la comprensione dei vari campi.

2.3 Flusso di generazione web services REST

Il flusso di generazione di un web services REST è il seguente:

1. Generazione codice del server.
2. Generazione codice del client.
3. Modifica del server per inserire la business logic che utilizza le richieste.
4. Creazione di una classe Java per interrogare il server.

2.3.1 Generazione del codice

Il primo passo da svolgere è la generazione del client e del server tramite Swagger Editor. Aprire il seguente link:

```
https://editor.swagger.io/
```

Una volta caricata l'interfaccia del sito bisogna importare il file contenente la descrizione della nostra calcolatrice. Quindi:

1. File → Import file.
2. Selezionare il file `calcolatrice.yaml`.

Se tutto è fatto correttamente nel browser a destra si può osservare come in Figura 2.1.

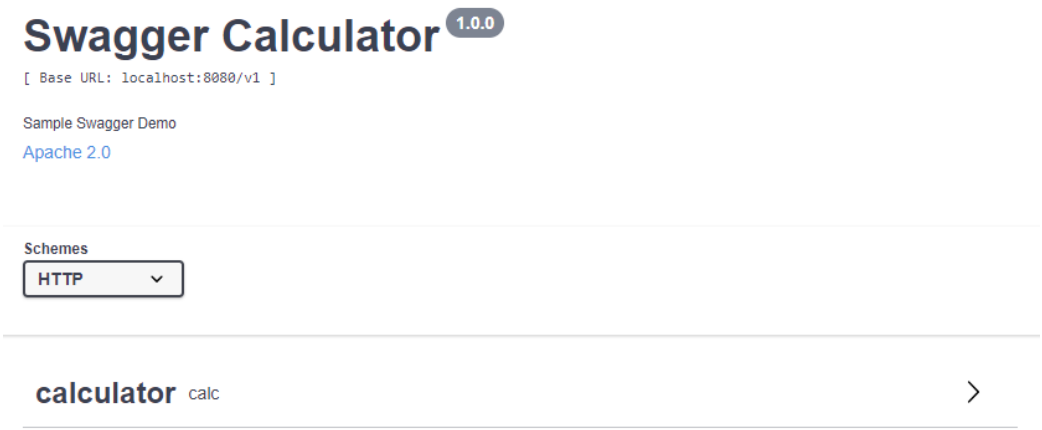

Figura 2.1: Risultato in Swagger Editor

Ora, possiamo generare il codice del server e client nel seguente modo:

- **server**:
  1. Scaricare il codice con: Generate Server → spring.
  2. Scaricato il file `spring-server-generated.zip`, estrarre i file all'interno.
- **client**:
  1. Scaricare il codice con: Generate Client → java.
  2. Scaricato il file `java-client-generated.zip`, estrarre i file all'interno.

2.3.2 Configurazione del server

Una volta ottenuto il codice del server, aprire Eclipse. La prima cosa che dobbiamo fare è importare il progetto contenente il codice del server all'interno di Eclipse.

1. File → import...
2. Selezionare Maven → Existing Maven Projects.
3. Selezionare la cartella contenente il codice del server.
4. Premere Finish.

Importare il progetto potrebbe richiedere alcuni minuti in quanto Eclipse potrebbe dover scaricare componenti aggiuntivi; lo stato dell'import può essere osservato in basso a destra. Una volta importato il progetto:

1. Selezionare il progetto **swagger-spring**.
2. Tasto destro → Run As → Maven build...
3. Nella nuova finestra inserire in **Goals**:

   ```
   clean package
   ```

   dove **clean** rimuove file generati durante la build e **package** prende il codice compilato e lo "impacchetta" all'interno del formato di distribuzione (ad esempio JAR).
4. Premere Run.

Il comando **Run** potrebbe richiedere alcuni minuti in quanto Maven deve scaricare le librerie necessarie al server e al termine riavviare Eclipse. Dopo il riavvio, la cartella `swagger-spring` ha la struttura illustrata in Figura 2.2.

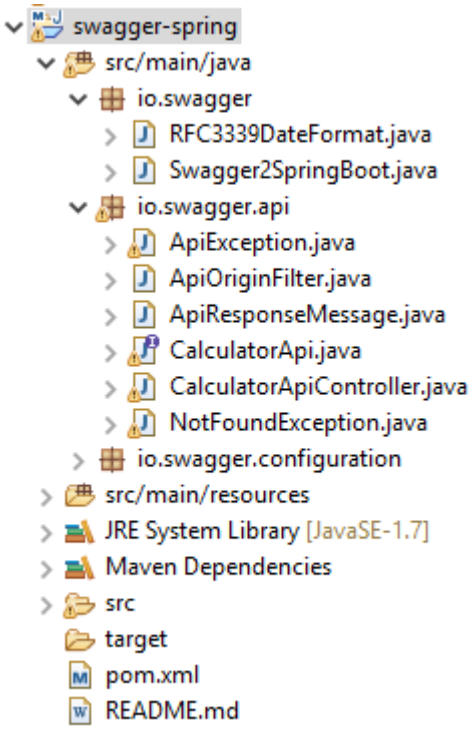

Figura 2.2: Cartella del codice del server

Per lanciare il server:

1. `swagger-spring` → `src/main/java` → `io.swagger`
2. lanciare `Swagger2SpringBoot.java` che configura tutto e fa partire il server contenente la API della calcolatrice.

Si noti che il codice eseguito contiene anche un server HTTP in ascolto su localhost alla porta 8080 (la porta su cui stare in ascolto è indicata nel file .yaml). Una volta lanciato il server si può aprire Swagger UI per visualizzare le API collegandosi al seguente indirizzo:

```
http://localhost:8080/v1/swagger-ui.html
```

si ottiene la pagina illustrata in Figura 2.3 nella quale si possono anche inserire i parametri ed inviare la richiesta al server.

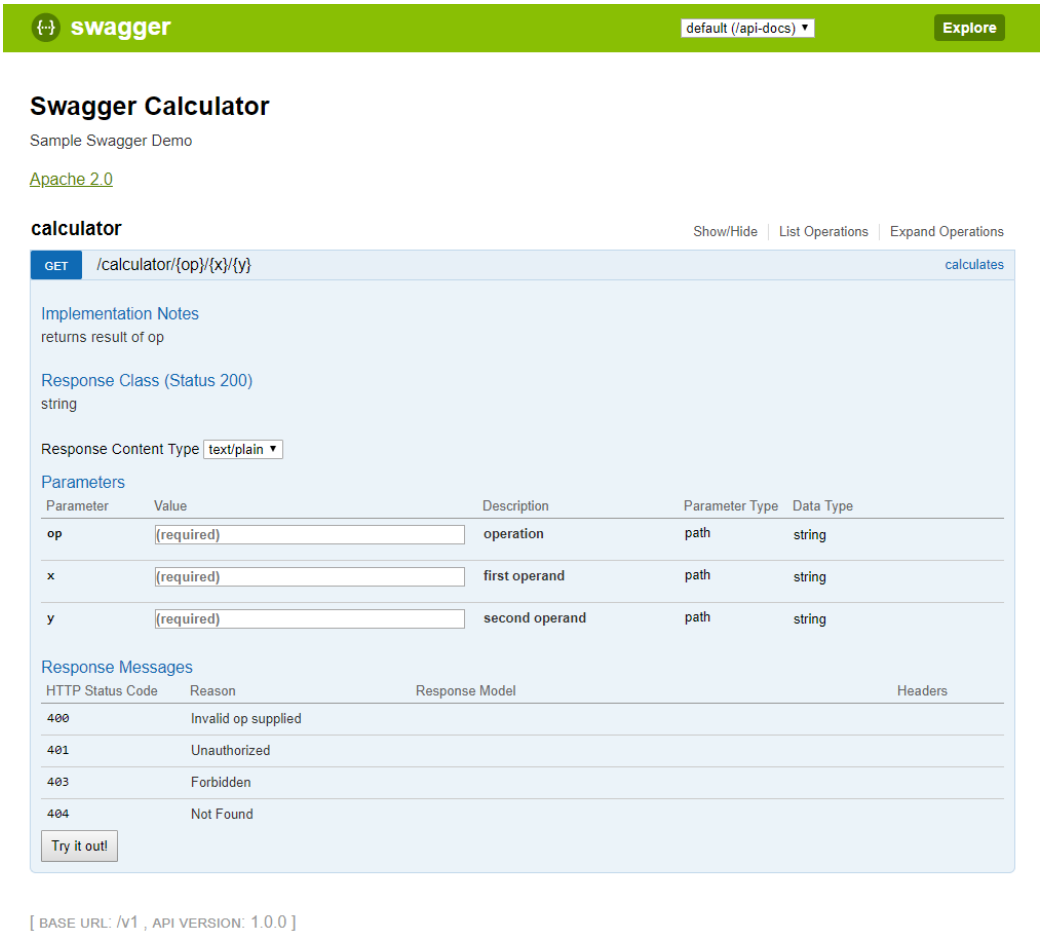

Figura 2.3: Visualizzazione dell'API della calcolatrice in Swagger UI

2.3.3 Configurazione del client

Per quanto riguarda l'import del client, eseguire le stesse operazioni mostrate nel Capitolo 2.3.2. Se il comando Run produce `BUILD FAILURE` allora, nelle impostazioni, bisogna cambiare da JRE a JDK. Di seguito è mostrato come fare con Eclipse per Windows:

1. Windows → Preferences.
2. Java → Installed JREs.
3. Selezionare la JRE presente nella lista e cliccare su Edit.
4. Cliccare su Directory...

5. Nella nuova finestra entrare nella cartella che contiene la JDK e selezionare tale cartella.

6. Finish → Apply and Close.

7. Premere nuovamente Run.

Una volta terminata l'esecuzione del comando **Run** la cartella `swagger-java-client` ha la struttura illustrata in Figura 2.4.

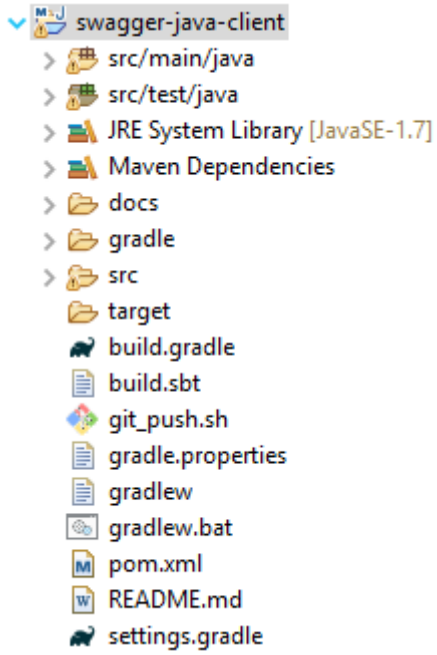

Figura 2.4: Cartella del codice del client

Ora, aggiungiamo una classe Java che ci permette di interrogare il server che contiene la calcolatrice.

1. `swagger-java-client` → `src/main/java`

2. Creare una classe Java chiamandola `CalculatorApiExample`.

3. Aprire il file `README.md` e copiare il codice fornito come esempio nella classe generata al punto precedente.

Ora possiamo lanciare la classe `CalculatorApiExample` per fare richieste al server, ovviamente deve essere modificata per inviare stringhe contenenti:

- **operazione**: somma/sottrazione/moltiplicazione/divisione.

- **primo operando**.
- **secondo operando**.

Il codice del client è generato automaticamente in modo da puntare al server in ascolto su *localhost:8080* perché così è scritto nel file .yaml. In uno scenario reale, su tale file devono essere indicati nome (o indirizzo IP) e porta della macchina su cui gira il web service.

Capitolo 3

Esercizi

3.1 Esercizio 1

Swagger ha generato il codice del server ma non ha generato anche il codice che implementa la funzionalità della calcolatrice (business logic). In questo esercizio occorre implementare il codice relativo alla calcolatrice cioè la parte di codice in cui, ricevuti i parametri in input, occorre fare la somma/sottrazione/moltiplicazione/divisione degli operandi. Riprendiamo la Figura 2.2 che illustra la struttura della cartella contenente il codice del server e concentriamoci sulla classe `CalculatorApiController` che contiene il metodo `compute()` richiamato ogniqualvolta noi mandiamo una richiesta di calcolo alla nostra calcolatrice. Il nome del metodo è stato definito a priori nel file `calcolatrice.yaml`. Questo metodo presenta la seguente struttura di default:

```
public ResponseEntity<String> compute(
@ApiParam(value = "operation",required=true) @PathVariable("op") String op,
@ApiParam(value = "first operand",required=true) @PathVariable("x") String x,
@ApiParam(value = "second operand",required=true) @PathVariable("y") String y)
{
    String accept = request.getHeader("Accept");
    if (accept != null && accept.contains("")) {
        try {
            return new ResponseEntity<String>(objectMapper.readValue(
                "", String.class),
                HttpStatus.NOT_IMPLEMENTED);
        } catch (IOException e) {
            log.error("Couldn't serialize response for content type ", e);
            return new ResponseEntity<String>(
                HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    return new ResponseEntity<String>(HttpStatus.NOT_IMPLEMENTED);
}
```

Il metodo ritorna un oggetto di tipo `ResponseEntity<String>` che oltre a contenere la stringa del risultato contiene anche il **response code** della richiesta ricevuta. Una volta modificato il metodo `compute()` verificate la correttezza dell'implementazione della calcolatrice testando sia tramite browser web che tramite il client.

3.2 Esercizio 2

In questo esercizio dovete provare ad invocare il servizio in esecuzione su un altro PC in cui sta girando il server. Visto che in questa fase cambia l'URL (non è più `localhost:8080`) a cui vi dovete collegare, è necessaria una piccola modifica nel file `calcolatrice.yaml`. Modificate il file, eseguite nuovamente le istruzioni relative al client indicate nel Capitolo 2.3.3 e inviate una richiesta di calcolo al server che gira sul PC di un vostro collega.

3.3 Esercizio 3

In questo esercizio è richiesta la modifica della calcolatrice all'insaputa dell'altro studente che sta invocando il servizio da un altro PC. La modifica consiste nell'aggiungere una costante K al risultato finale dell'operazione. Lo studente che esegue client deve indovinare la costante K. Questo esercizio serve a dimostrare che l'approccio Web Service permette di modificare una business logic interna senza dover modificare il codice di tutti i client che trovano subito automaticamente (e inconsapevolmente) la nuova funzionalità.