POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Generazione automatica di test per servizi REST

Relatore
prof. Antonio Lioy

Candidato
Leandro Battaglia

Anno accademico 2017-2018

Sommario

L'obiettivo del presente elaborato è presentare una applicazione che può essere impiegata per effettuare test di servizi web.

La prima parte dell'elaborato cerca di illustrare i concetti principali su cui si fonda l'architettura REST e i linguaggi che sono stati proposti per descrivere i servizi web, ponendo una particolare attenzione su Swagger/OpenAPI che nel corso degli ultimi anni è stato adottato su larga scala per descrivere, documentare e consumare servizi web. Punto di partenza, infatti, per l'applicazione assemblata è la descrizione Swagger del servizio web che si vuole testare.

La parte successiva, consiste in una indagine nel mondo del software libero in cui si è andati alla ricerca di tool che potessero essere utili a creare un'applicazione più complessa in grado di testare nel modo più automatizzato possibile un servizio web. Nel corso di tale ricerca, tre tool si sono rivelati particolarmente utili nella creazione di un'applicazione simile. Si tratta dei moduli swagger-test-template, open-api-test-generator e json-schema-test-data-generator, tutti disponibili in maniera aperta su Github.

Durante la realizzazione dell'applicazione tali moduli sono stati opportunamente modificati, al fine di ottenere un comportamento quanto più vicino a quello desiderato nella fase iniziale.

La parte centrale dell'elaborato è dedicata alla spiegazione delle caratteristiche dell'applicazione e su come i moduli interagiscano tra di loro. Inoltre, l'intera esposizione riporta i vari problemi e difficoltà che si sono incontrati nel corso dello sviluppo, le possibili alternative ed infine la soluzione che è stata adottata. L'intera applicazione consiste in una serie di moduli JavaScript che, a partire dalla descrizione Swagger del servizio che si vuole testare, generano una serie di unit test. Ognuna di queste unit test generate è relativa a uno specifico endpoint esposto dal servizio. Il modulo principale, a partire dalle informazioni contenute nella descrizione, per ogni singolo endpoint, grazie al contributo di un generatore casuale che si occupa di creare i dati da assegnare ai vari parametri previsti per le richieste, genera delle unit test che contengono delle asserzioni. Le asserzioni sono relative al codice di stato della risposta e al contenuto della risposta del servizio. La generazione delle unità di test passa attraverso l'utilizzo di una serie di template, opportunamente configurati, che consentono la generazione automatica di codice che può essere immediatamente eseguito.

Infine l'ultima parte è dedicata all'analisi dei risultati che si sono ottenuti provando l'applicazione su servizi preesistenti o che sono stati implementati appositamente per testare l'applicazione.

Ringraziamenti

Il lavoro descritto in questo elaborato è stato svolto sotto la supervisione del Prof. Antonio Lioy che ha seguito con attenzione gli sviluppi della tesi.

Un ringraziamento speciale va inoltre al responsabile architetture e tecnologie del CSI Piemonte il dottor Enzo Veiluva e ai suoi collaboratori il dottor Marco Boz, il dottor Egidio Bosio e il dottor Francesco Giurlanda.

Indice

Capitolo 1

Servizi REST e linguaggi di descrizione

In questo capitolo verranno presentati alcuni tra gli approcci e i linguaggi di descrizione per servizi web più diffusi cercando di confrontarli e metterne in luce pregi e difetti.

Secondo il W3C un servizio web è un sistema software progettato per supportare l'interoperabilità tra diverse macchine che interagiscono nella stessa rete [1]. Le tecnologie impiegate in tale architettura sono XML, SOAP e WSDL. Tuttavia nel 2004, il W3C ha esteso la definizione di servizio web includendo un nuovo stile architetturale chiamato REST (REpresentational State Transfer) proposto da Roy Fielding e basato sul protocollo HTTP. Più in generale è basato su un protocollo che è:

- client-server;
- privo di stato;
- a livelli e cacheable;
- dotato di un'interfaccia uniforme.

Un concetto chiave in REST è il concetto di risorsa, ovvero informazioni identificate da un URI. Le risorse non vengono mai trasferite. Ciò che viene trasferita è una rappresentazione delle risorsa stessa e i formati utilizzati sono solitamente XML e JSON. Tuttavia le risorse non sono statiche. I client, infatti, possono manipolare le risorse di un servizio sfruttando le operazioni ammesse dal servizio stesso. Le operazioni sulle risorse di un servizio REST sono mappate sui metodi HTTP secondo lo schema CRUD (Create, Read, Update, Delete).

Operazione CRUD	*Metodo HTTP*
Create	POST
Read	GET
Update	PUT, PATCH
Delete	DELETE

Tabella 1.1. Le operazioni CRUD e i corrispondenti metodi HTTP.

Se si volesse, per esempio, implementare un servizio web, seguendo i vincoli dell'architettura REST, per gestire dei profili di utenti, si potrebbe usare lo schema proposto nella Tab. 1.2.

Dalla Tab. 1.2 si vede come un servizio che segue i principi REST è un servizio che espone risorse e collezioni di risorse, a differenza di quanto avviene in un servizio SOAP in cui gli URL esposti corrispondono ad azioni.

Richiesta	*Descrizione*	*Codice HTTP*
GET */users*	Ritorna l'intera collezione di utenti	200
GET */users/{userId}*	Ritorna l'utente con id *userId* se presente	200, 404
POST */users*	Crea un nuovo utente e lo aggiunge alla collezione	201, 403, 409
PUT */users*	Aggiorna l'intera collezione	200, 204
PUT */users/{userId}*	Aggiorna l'utente con id *userId*	200, 204
DELETE */users*	Elimina l'intera collezione di utenti	200, 202, 204
DELETE */users/{userId}*	Elimina l'utente con id *userId*	200, 202, 204

Tabella 1.2. Le richieste HTTP e alcuni possibili codici di stato del servizio per i profili di utenti.

1.1 HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) è un vincolo per applicazioni basate su architettura REST [2]. Secondo Roy Fielding si tratta di un requisito essenziale affinché un servizio web possa essere classificato come API REST:

> What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. [3]

L'idea di base è cercare di mettere sullo stesso piano servizi web e applicazioni web. L'unica differenza è che i servizi sono pensati per essere consumati da macchine, mentre le applicazioni web sono per gli umani. Ma a parte questo il modo in cui un servizio REST dovrebbe funzionare, seguendo questo approccio, è simile a quello di un'applicazione web.

Un servizio che rispetta il vincolo HATEOAS è un servizio in cui:

- le interazioni sono prive di stato;
- le informazioni di stato sono costituite da hyperlinks;
- i cambi di stato avvengono seguendo gli hyperlinks;
- l'applicazione è auto-descrittiva.

Ogni richiesta inviata a un servizio di questo tipo contiene tutto ciò che serve affinché venga processata, e non necessita di alcun riferimento a interazioni passate.

Le informazioni di stato sono hyperlinks che vengono inclusi nelle risposte del servizio. Quando il servizio risponde include nella rappresentazione della risorsa richiesta un elenco di link, possibilmente accompagnati da dettagli aggiuntivi, a cui il client può inviare nuove richieste. Da qui il concetto di interazioni REST guidate da hypermedia. Al client, noto l'hyperlink, non resta che scoprire i metodi supportati per ogni nuovo endpoint e per farlo può sfruttare il metodo HTTP OPTIONS. Da questi primi concetti deriva una conseguenza importante: è il client a scegliere la sequenza di link da seguire realizzando così una sessione. Inoltre il client non costruisce autonomamente gli URL ma invia richieste scegliendo tra quelli restituiti dal servizio in ogni risposta. In questo modo il client deve solo conoscere l'URL della risorsa principale.

Infine le applicazioni che seguono questo approccio vengono definite auto-descrittive in quanto sono i client a scoprire quello che offre il servizio usandolo, proprio come quando si visita un sito web. Il client quindi può "esplorare" il servizio autonomamente senza disporre di una descrizione del servizio stesso.

La scelta di non esporre alcuna descrizione rende i servizi basati su HATEOAS estremamente orientati ad essere consumati da umani e non da macchine. Pertanto, tenendo a mente quanto detto sopra, per un client risulta impossibile conoscere a priori quali sono le risorse esposte dal server e

le loro rappresentazioni. Tant'è che le applicazioni REST che rispettano il vincolo HATEOAS sono applicazioni web spesso dotate di pagine HTML che guidano il client a consumare il servizio. In questo modo anche se il client, per esempio, esegue una POST a un certo URL non avrà bisogno di conoscere la struttura dei dati da inserire nel corpo della richiesta poiché sarà aiutato da un form web.

È comunque importante ricordare che le applicazioni che seguono il vincolo HATEOAS sono applicazioni che possiedono il più alto livello di maturità nella scala proposta da Leonard Richardson in occasione del QCon di San Francisco [4]. Si tratta di un modello di maturità RESTful per classificare i servizi web, noto come Richardson Maturity Model. Lo scopo del modello è quello di classificare i servizi a secondo di quanto sono conformi ai vincoli dell'architettura REST. Il modello include quattro livelli:

- Livello 0 (no REST): HTTP è usato come "tunnel" ed è disponibile una singola risorsa;
- Livello 1 (Risorse): il servizio espone più di una risorsa;
- Livello 2 (HTTP verbs): è possibile effettuare operazioni sulle risorse sfruttando più di un metodo HTTP;
- Livello 3 (Hypermedia controls): le risposte includono hyperlinks e il servizio è auto-descrittivo.

1.2 Linguaggi di descrizioni per servizi web

Generalmente i servizi web sono auto-descrittivi. Ciò vuol dire che il servizio espone un certo tipo di descrizione che può essere impiegata per capire come consumare il servizio stesso. Tali descrizioni utilizzano linguaggi che appartengono a una famiglia ben più ampia nota con il nome di IDL (Interface Description Language).

Un linguaggio di descrizione per servizi web è un linguaggio formale che consente di creare una descrizione del servizio. In generale, la descrizione di un servizio web svolge la funzione di contratto tra il fornitore del servizio e i suoi consumatori. Tali descrizioni strutturate permettono di conoscere quali sono le interfacce esposte e quali sono i formati delle risorse disponibili. Inoltre le descrizioni dei servizi web possono essere utilizzate per generare documentazione e codice in linguaggi di programmazione diversi.

Per servizi web standard il linguaggio di descrizione di riferimento è il WSDL (Web Services Description Language). Il WSDL è un linguaggio basato su XML e universalmente impiegato per descrivere servizi SOAP. Ad oggi, nell'ambito dei servizi web REST, non esiste ancora un linguaggio di descrizione standard e accettato su scala globale. Tuttavia, negli ultimi anni sono state proposte diverse soluzioni.

1.3 WADL

Il WADL è un linguaggio di descrizione progettato per creare descrizioni machine-readable di servizi basati su HTTP [5]. Si tratta di un linguaggio basato su XML.

Attraverso il WADL è possibile:

- descrivere le risorse esposte dal servizio;
- specificare le relazioni tra le risorse;
- specificare i metodi HTTP ammessi per ogni risorsa e i relativi parametri;
- definire i formati delle rappresentazioni delle risorse.

Una descrizione WADL è costituita da un unico elemento `<application>` contenente:

- zero o più elementi `<doc>`;
- un elemento `<grammar>` opzionale;
- zero o più elementi `<resources>`;
- zero o più:
  - elementi `<resource_type>`;
  - elementi `<method>`;
  - elementi `<representation>`;
  - elementi `<param>`.

In Fig. 1.1 una descrizione semplificata scritta in WADL del servizio che gestisce utenti.

Tra tutti ci soffermeremo in particolare sugli elementi `<grammar>` e `<representation>`. L'elemento `<grammar>` funge da contenitore per le definizioni dei tipi di dato usati per la rappresentazione delle risorse. Le definizioni possono essere incluse per riferimento e sono espresse sfruttando lo schema XML. L'elemento `<representation>` può essere contenuto all'interno degli elementi `<request>` e `<response>` e contiene un attributo chiamato *element* che specifica il qualified name di un root element dichiarato nella sezione `<grammar>`. Inoltre, il tag `<representation>` include un altro attributo chiamato *mediaType* che indica il media type supportato per la rappresentazione della corrispondente risorsa. Alcuni tra i media type più comuni sono *application/json*, *application/xml* e *text/plain*. Ciò vuol dire che è possibile costruire servizi, descritti in WADL, che trattano rappresentazioni di risorse utilizzando il formato JSON. Il che renderebbe WADL un buon candidato nell'ambito della scelta di un linguaggio di descrizione per servizi web tenendo a mente l'obiettivo finale della tesi. Tuttavia, come già specificato in precedenza, WADL è un linguaggio basato su XML. Pertanto non è possibile fare in modo che gli schemi JSON delle rappresentazioni delle risorse vengano esposti dal servizio o in qualche modo specificati all'interno di una descrizione WADL. Quello che è possibile ottenere quando si impiegano schemi JSON lato server è che il servizio esponga nella sezione `<grammar>` del WADL un riferimento a uno schema XML che è la traduzione degli schemi JSON in elementi XML.

Un'alternativa a WADL è il WSDL 2.0. WSDL 2.0 è un linguaggio di descrizione basato su XML per descrivere servizi web di tipo REST, reso standard, come "raccomandazione", dal W3C. Si tratta di una versione successiva al WSDL 1.1, già usato per descrivere servizi SOAP ma mancante del supporto per alcuni metodi HTTP ad eccezione dei metodi GET e POST. Tuttavia WSDL 2.0 risulta al quanto limitato in quanto non consente di specificare le risorse come invece e possibile fare con WADL.

Infine, entrambi WADL e WSDL vengono solitamente impiegati per generare descrizioni a "posteriori", ovvero a partire dal codice sorgente del servizio web completo.

1.4 OpenAPI, RAML e API Blueprint

Negli ultimi anni, nell'ambito dei linguaggi di descrizioni per API HTTP i tre rivali principali sono:

- la specifica OpenAPI (nota anche con il nome di Swagger), di Open API Initiative;
- RAML, di MuleSoft;
- API Blueprint, di Apiary.

Ad oggi, tra i tre concorrenti, la specifica OpenAPI sembra essere la più famosa e la più diffusa. Tra le aziende più famose a supporto della OpenAPI Initiative ci sono Adobe, IBM, SAP, PayPal, Atlassian, Google e Microsoft.

```
<doc xmlns:jersey="http://jersey.java.net/"/>
<grammars>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
    <xs:element name="user"/>
      <xs:complexType name="user">
        <xs:sequence>
          <xs:element name="id" type="xs:int" minOccurs="1"/>
          <xs:element name="username" type="xs:string" minOccurs="1"/>
       </xs:sequence>
     </xs:complexType>
    </xs:schema>
</grammars>
<resources base="http://localhost:8080/test_service/rest/">
  <resource path="">
    <resource path="users/{usersId}">
      <method id="get_user" name="GET">
        <request>
          <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="usersId"
              style="query" type="xs:int"/>
       </request>
       <response>
          <representation mediaType="application/json" element="user"/>
       </response>
      </method>
    </resource>
    <resource path="users">
      <method id="post_user" name="POST">
        <request>
          <representation xmlns="http://wadl.dev.java.net/2009/02"
              element="user" mediaType="application/json"/>
        </request>
        <response>
          <representation xmlns="http://wadl.dev.java.net/2009/02"
              element="user" mediaType="application/json"/>
        </response>
      </method>
    </resource>
  </resource>
</resources>
</application>
```

Figura 1.1. Esempio di descrizione WADL.

Tra le tre, Swagger è la meno giovane (primo commit nel luglio 2011), mentre RAML e API Blueprint vennero create due anni dopo. Tutte e tre le specifiche possono essere utilizzate per generare documentazione interattiva e per generare codice lato server e lato client.

Nel 2016 Apiary, adesso parte di Oracle, si è unita alla OpenAPI initiative. MuleSoft, i creatori di RAML, ha fatto lo stesso nell'aprile 2017, facendo in modo che la specifica OpenAPI diventasse il formato di riferimento nel panorama dei linguaggi di descrizione per API REST. L'unione di MuleSoft e Apiary alla OpenAPI Initiative ha fatto sì che entrambi aggiungessero nei propri tool il supporto per le specifiche espresse in Swagger, e ha consolidato l'importanza della specifica OpenAPI. Nel frattempo sono stati sviluppati diversi tool di conversione per tradurre descrizioni da una specifica a un'altra. Ad oggi resta solo da capire come RAML e API Blueprint contribuiranno

ad estendere Swagger.

Nel corso dei paragrafi successivi si cercherà di mettere in luce gli aspetti principali di ognuna delle tre specifiche.

1.4.1 La specifica OpenAPI

La specifica OpenAPI [6] è una specifica aperta, basata sul contributo di una community e facente parte della OpenAPI initiative, progetto della Linux Foundation. Si tratta di una specifica, a volte nota anche con il nome di Swagger, impiegata per realizzare descrizioni e documentazioni per servizi web. La specifica OpenAPI consente di definire descrizioni indipendenti dal linguaggio di programmazione. A partire dalle descrizioni è possibile ottenere documentazioni interattive e generare codice per server, client e test automatici. Le descrizioni sono scritte usando due formati:

- JSON;
- YAML. [1]

Inoltre è possibile ottenere la descrizione a partire dal codice sorgente di un servizio.

Uno dei software open-source più diffuso e impiegato per progettare e documentare servizi seguendo la specifica OpenAPI è Swagger, sviluppato da SmartBear Software. SmartBear mette a disposizione una serie di tool come Swagger Editor, Swagger Codegen e Swagger UI rispettivamente per progettare, generare codice e documentare API. Tutte queste funzioni sono disponibili contemporaneamente su una piattaforma chiamata SWAGGERhub [7].

In Fig. 1.2 una descrizione semplificata scritta in OpenAPI 3.0 del servizio che gestisce utenti.

Ogni descrizione OpenAPI si apre con una sezione dedicata a metadati. Qui si specificano alcune informazioni sull'API (`title`, `description`, `version`, `termsOfService`, ...).

La sezione `server` elenca gli URL base per l'API. Tutti i path che appaiono nella sezione successiva sono relativi all'URL base qui specificato.

La sezione `paths` definisce gli endpoint dell'API. Per ogni endpoint sono specificati i metodi HTTP supportati. Ogni operazione include:

- una sezione `parameters` opzionale;
- una sezione `requestBody` opzionale;
- una sezione `Responses`.

Nella sezione `parameters` è possibile specificare i parametri passati via URL, query, header o cookie. Per ogni parametro si specificano il tipo, il formato, se è opzionale o richiesto e altri dettagli.

Per le operazioni in cui i client inviano dati nel corpo della richiesta (POST, PUT, PATCH), la sezione `requestBody` permette di specificare informazioni sulla rappresentazione della risorsa. In particolare, tramite l'oggetto `content` è possibile specificare i mediatype accettati dall'operazione e il relativo schema. OpenAPI supporta sia JSON che XML. Per gestire gli schemi più comodamente, OpenAPI offre la possibilità di definirli in una sezione dedicata della descrizione e utilizzare la notazione `$ref` per far riferimento alla definizione.

La sezione `responses` permette di specificare quali sono i possibili codici di stato (200, 404, 403, ...) e per ogni codice lo schema del corpo della risposta.

[1] "YAML Ain't a Markup Language" (originariamente "Yet Another Markup Language")

```
info:
  title: Sample API
  version: "1.0.0"
servers:
  - url: http://api.example.com/v1
paths:
  /users:
    post:
      requestBody:
        required: true
        content:
          'application/json':
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: Return the JSON created user
          content:
            'application/json':
              schema:
                $ref: '#/components/schemas/User'
  /users/{userId}:
    get:
      parameters:
        - name: userId
          in: path
          required: true
          description: The id of the user to retrieve
          schema:
            type: integer
      responses:
        '200':
          description: Return the JSON user
          content:
            'application/json':
              schema:
                $ref: '#/components/schemas/User'
        '404':
          description: user not found
components:
  schemas:
    User:
      required:
        - id
        - username
      properties:
        id:
          type: integer
        username:
          type: string
```

Figura 1.2. Esempio di descrizione OpenAPI 3.0 scritta in YAML.

1.4.2 RAML

RAML (RESTful API Modeling Language) è un linguaggio human e machine-readable per descrivere API REST basato su YAML [8]. Sebbene sia classificato come linguaggio di descrizione per API REST, risulta più corretto definirlo linguaggio di descrizione per API "practically RESTful", poiché è in grado di descrivere servizi che rispettano solo alcuni dei vincoli richiesti affinché un API possa essere definita completamente conforme all'architettura REST.

L'obiettivo di RAML è fornire un modo per descrivere le risorse, i metodi, i parametri, le risposte e le rappresentazioni di un servizio web. Si tratta di una specifica aperta non proprietaria, che permette anche di generare codice sorgente per client e server e documentare l'API.

La Fig. 1.3 mostra una descrizione RAML del servizio utenti.

```
#%RAML 1.0
title: Users API # required title
baseUri: http://example.api.com/
version: v1
types: # global type definitions that can be reused throughout this API
  User: # define type named User
    type: object
    properties:
      id: integer
      username: string
/users: # optional resource
  post:
    body:
      application/json:
        type: User
    responses:
      201:
        body:
          application/json:
            type: User
  /{userId}:
    uriParameters:
      userId:
        type: integer
    get:
      responses:
        200:
          body:
            application/json:
              schema: User
        404:
          description: user not found
```

Figura 1.3. Esempio di descrizione RAML

La sezione iniziale di un documento RAML include alcune informazioni di base sull'API, come `title`, `baseUri` e `version`. In questa sezione è anche possibile definire tipi e *traits*, ovvero attributi a livello di operazione che possono essere riusati in più metodi. Il sistema dei tipi presenta alcune caratteristiche della programmazione ad oggetti e degli schemi JSON e XML. È comunque possibile includere schemi JSON o XML, senza dover definire nuovi tipi usando il sistema di RAML.

La sezione `methods` include le operazioni eseguibili su ogni risorsa. Per ogni operazione è possibile specificare gli header attesi, i parametri accettati, lo schema previsto per il body della richiesta HTTP e le possibili risposte HTTP all'invocazione dell'operazione sulla risorsa. Ogni risposta contiene i possibili codici di stato HTTP e, per ogni codice di stato si specifica solitamente la struttura del body.

In base a quanto visto finora risulta evidente come OpenAPI e RAML siano molto simili.

1.4.3 API Blueprint

API Blueprint è un linguaggio per progettare API web [9]. Utilizza il linguaggio Markdown ed è conforme alla sintassi Flavored Markdown di Github. Un documento blueprint è strutturato in sezioni logiche. Ogni sezione occupa una posizione nel documento ed è dedicata a contenuti specifici. Tutte le sezioni sono opzionali, ma se presenti devono seguire una struttura ben precisa. La Fig. 1.4 mostra una descrizione Blueprint del servizio utenti.

```
FORMAT: 1A
HOST: http://api.example.com/
# user api
Simple api for users
## Users Collection [/users]
### Create a New User [POST]
It takes a JSON object containing
a user and return the created user.
+ Request (application/json)
        {
            "id": "1234",
            "username": "user1"
        }
+ Response 201 (application/json)
    + Body
            {
                "id": "1234",
                "username": "user1"
            }
## Get a single user [/users/{id}]
+ Parameters
    + id: 123 (number) - An unique identifier of the user.
### Retrieve a User [GET]
+ Response 200 (text/plain)
    + Body
            {
                "id": "1234",
                "username": "user1"
            }
```

Figura 1.4. Esempio di descrizione API Blueprint

Le sezioni possibili sono:

- Metadata;
- API name & overview;
- Resource;

- Resource Group;
- Data Structures.

La sezione `resource` è la più importante in quanto specifica l'insieme di risorse disponibili a uno specifico URL. Qui le due sezioni più importanti sono la sezione `attributes` e la sezione `model`. Nella sezione `attributes` si descrive lo schema usato per rappresentare le risorse. Le strutture dati sono definite usando il formato MSON (Markdown Syntax for Object Notation), un formato, sviluppato da Apiary, per descrivere schemi che siano human e machine readable. La sezione `model` fornisce informazioni sugli header e i mediatype accettati. La sezione `action` descrive le caratteristiche delle richieste e delle risposte HTTP.

L'ultima parte del documento è solitamente riservata alle strutture dati che, anche in questo caso sono descritte con il formato MSON.

API Blueprint segue pressoché lo stesso disegno proposto da Swagger e RAML. Le differenze principali stanno nel formato scelto (Markdown anziché JSON o YAML) e nel formato per le rappresentazioni (MSON anziché JSON o XML).

Capitolo 2

Progettazione

Il primo passo che è stato fatto al fine di implementare un ambiente in grado di testare automaticamente servizi web è consistito nell'analizzare alcuni degli strumenti open source disponibili in rete. In questo capitolo verranno dunque presentati i tool esistenti che sono stati presi in considerazione per la creazione dell'ambiente di test.

2.1 Software proprietari

I primi strumenti che verranno presentati sono software proprietari. Si tratta di soluzioni più o meno professionali a cui è possibile accedere acquistando una licenza o un abbonamento. Questi programmi, disponibili gratuitamente solo in versione trial per un periodo limitato, mettono a disposizione un'elevata quantità di funzionalità oltre a quella dell'automated REST API testing. Tali strumenti non sono stati provati direttamente, in quanto non open source, ma solo analizzati tramite l'ausilio di documentazione, guide e video dimostrativi online.

2.1.1 vREST

La funzionalità principale di vREST consiste nella creazione ed esecuzione di *test cases*. In ogni *test cases* l'utente specifica:

- i dettagli della richiesta HTTP (URL, metodo ed eventuali parametri);
- le validazioni desiderate (asserzioni sui campi e lo schema della risposta).

Quando i test vengono eseguiti, il software comunica l'esito di ogni test. I test producono un esito positivo se tutte le asserzioni sono verificate, altrimenti i test falliscono. vREST offre la possibilità di definire variabili, proprio come in un linguaggio di programmazione. Ad ogni variabile è associato un valore e può essere impiegata per richieste e validazioni. Inoltre è possibile estrarre variabili da risposte precedenti e riusarle in test successivi.

vREST, infine, include un modulo in grado di effettuare *Record and Replay* testing. Anche nota col nome di *Record and Playback*, si tratta di una tecnica di testing che consente di registrare operazioni manuali (record) e riprodurle automaticamente in qualsiasi momento (replay). Nel caso specifico di test per servizi web, le operazioni coincidono con chiamate HTTP indirizzate a uno specifico URL e caratterizzate da un metodo e parametri aggiuntivi. In vREST è possibile analizzare e modificare i *test cases* generati in modo da renderli più flessibili attraverso l'uso delle variabili e delle validazioni. vREST rimane un ottimo tool per testare servizi web, tuttavia lo svantaggio principale è costituito dal fatto che i test vanno scritti a mano per ogni caso e non è previsto alcuno strumento per generare unità di test a partire da una descrizione del servizio. Il modulo di *Record and Replay* risulta comodo solo per applicazioni che cambiano raramente. Se le applicazioni sono continuamente aggiornate è necessario registrare nuovamente i test.

2.1.2 Runscope

Runscope è un programma che offre soluzioni per monitorare, testare e debuggare applicazioni web. La funzionalità principale di Runscope è l'API monitoring, ovvero il monitoraggio del comportamento dell'applicazione in funzione. Dal punto di vista del functional testing, Runscope, è molto simile a vREST. Anche qui è possibile definire test manualmente. Tuttavia, Runscope, offre la possibilità di generare automaticamente i test a partire da una descrizione del servizio web espressa in Swagger/OpenAPI. I test così generati risultano però mancanti di eventuali parametri di input che devono essere inseriti manualmente.

2.1.3 SOATest

SOATest è un altro software proprietario impiegato per functional testing, security testing e anche load e performance testing. La caratteristica che lo rende, in questa trattazione, degno di nota è la possibilità di generare automaticamente test a partire da una descrizione espressa in WSDL. Anche qui, come in Runscope, sta all'utente completare le richieste con i parametri necessari.

2.1.4 Assertible

Assertible è l'ultimo dei software proprietari che verrà presentato. Si tratta di un tool che offre soluzioni gratuite per piccoli team e soluzioni a pagamento per progetti più grandi. Assertible racchiude quasi completamente tutte le caratteristiche dei software presentati finora. Infatti, è possibile:

- creare test manualmente per ogni endpoint esposto dal servizio, specificando i parametri della richiesta e le asserzioni desiderate;
- monitorare lo stato del servizio lanciando periodicamente i test creati e ricevere alert quando si verificano determinati eventi;
- importare le specifiche di un servizio descritto in Swagger/OpenAPI per creare automaticamente un test per ogni combinazione di endpoint-metodo HTTP.

Come si vede Assertible include le caratteristiche principali di vREST e di Runscope, tuttavia essendo un programma closed source non verrà impiegato nella creazione dell'ambiente di testing, ma come si vedrà più avanti, esistono molti tool open source che hanno caratteristiche simili a quelle presentate da Assertible.

2.2 Software open source

La prossima categoria di software presentata è quella dei tool open source. Si tratta di programmi aperti e disponibili su Github. Nella maggior parte dei casi sono privi di interfaccia grafica e impiegabili tramite riga di comando. Il linguaggio di programmazione principalmente adoperato è JavaScript.

Complessivamente, tra tutti i software open source che sono stati provati e che verranno presentati è possibile distinguere due classi principali, che sono già emerse nell'analisi dei software proprietari.

Alla prima categoria appartengono quei tool che sono esclusivamente costituiti da un client HTTP e un validator. Il client è impiegato per effettuare le chiamate HTTP al servizio da testare, mentre il validator mette a disposizione dei metodi per validare i contenuti delle risposte fornite dal servizio. A seconda dell'implementazione del client è possibile gestire o meno aspetti avanzati del protocollo, come cookie, header e autenticazione. Per quanto riguarda il validator, invece, si tratta di una libreria dotata di un'interfaccia di programmazione che cerca, spesso, di rendere

l'invocazione dei metodi molto simile al linguaggio umano. Se, per esempio, si vuole verificare che il codice di stato di una risposta HTTP, che chiameremo `resCode`, sia 200, l'istruzione corrispondente potrebbe essere `expect(resCode).to.be.equal(200)`, oppure `resCode.should.be(200)`. Come detto il validator viene usato per effettuare asserzioni sul contenuto delle risposte sia a livello di intestazione che a livello del body. Quello che comunque accomuna tutti questi tool è il fatto che i test vanno scritti manualmente, e l'aspetto automatizzato riguarda solo la verifica delle validazioni desiderate.

I programmi appartenenti alla seconda categoria sono molto simili a quelli della prima, ma includono un aspetto aggiuntivo che risulta essere rilevante, ovvero la capacità di generare codice automaticamente a partire da una descrizione del servizio web. Questi tool sono in grado di interpretare la descrizione di un servizio web fornita in ingresso, e produrre del codice, compilabile ed eseguibile, molto simile a quello producibile manualmente attraverso i programmi dell'altra classe. Ogni programma ha dei prerequisiti specifici che verranno spiegati successivamente, cercando così di capirne vantaggi e svantaggi.

2.2.1 Frisby.js

Frisby.js è un framework per testare API REST scritto in JavaScript e disponibile come package di Node.js. Utilizza lo stile di validazione di Jasmine, e Jest per eseguire i test creati. Frisby.js appartiene a quella classe di tool in cui i test vanno creati manualmente. In Fig. 2.1 viene mostrato un test minimale scritto usando il framework.

```
const frisby = require('frisby');

describe('new test suite',function(){
  it('should be a teapot', function (done) {
    frisby.get('http://httpbin.org/status/418')
      .expect('status', 418)
      .done(done);
  });
});
```

Figura 2.1. Esempio di test Frisby.JS.

Il test in Fig. 2.1 effettua una chiamata HTTP, usando il metodo GET, all'URL indicato. La pagina è un'implementazione di un pesce d'aprile dell'IETF pubblicato nel 1998 con l'RFC-2324, che descrive un protocollo per monitorare e controllare caffettiere. L'URL in questione ritorna il codice di stato HTTP 418 (418 - I'm a teapot) e un body contenente un ASCII art di una teiera. Al di là dell'aspetto umoristico, rappresentato dall'easter egg, l'esempio mostra come è possibile, usando Frisby.JS, testare facilmente un servizio web. Il codice dell'esempio introduce un nuovo test tramite la funzione `it` ed effettua un'unica asserzione tramite la funzione `expect`. Quest'ultima funzione verifica che il valore di `status`, ovvero il codice di stato della risposta HTTP, sia 418. Ovviamente, quello appena presentato è, come detto, un esempio minimale che effettua un'unica asserzione sull'header della risposta. Tuttavia Frisby.JS consente di eseguire asserzioni e operazioni più complicate.

Frisby.JS mette a disposizione dei metodi HTTP predefinite che corrispondono ai metodi GET, POST, PUT e DELETE.

Inoltre è possibile effettuare, all'interno di una test suite, più chiamate concatenate tramite la funzione `then`, sfruttando lo stile JavaScript delle Promise. In questo modo ogni richiesta può sfruttare i dati della risposta precedente.

In Frisby.JS sono disponibili due tipi di validatori:

- i built-In expect handler;
- gli handler di Jasmine.

I built-In expect handlers sono delle parole chiavi che consentono di accedere facilmente ai campi della risposta per verificare il valore o la struttura. Ad esempio, come già visto, attraverso l'handler `status` si accede al codice di stato della risposta HTTP, oppure l'handler `header` dà accesso ai campi dell'intestazione della risposta. È anche possibile definire degli handler personalizzati grazie alla funzione `addExpectHandler`. La funzione riceve tra i parametri l'identificativo da assegnare all'handler personalizzato, nel body si specificano poi le asserzioni dell'handler stesso. Infine è possibile utilizzare gli handler di Jasmine, un framework di testing per JavaScript che segue le regole del behavior-driven development (BDD). Qui le validazioni avvengono usando dei metodi la cui invocazione è molto simile al linguaggio umano, come `toBe`, `toBeCloseTo`, `toBeGreaterThan`, `ToBeNull`. Inoltre, Jasmine espone una serie di metodi per eseguire codice prima e dopo l'esecuzione di una certa unità di test o di tutte (`afterAll`, `afterEach`, `beforeAll`, `beforeEach`).

Entrambi i validatori consentono in ogni caso di eseguire asserzioni sui JSON contenuti nei corpi delle risposte HTTP. Le validazioni riguardano sia i valori che lo schema del JSON. In particolare, le validazioni sui tipi avvengono sfruttando una libreria interna a Frisby.JS chiamata Joi.

Frisby.JS risulta essere un ottimo tool che rende la creazione e l'esecuzione di test per servizi web facile ed intuitiva. Tuttavia, come già anticipato, non contiene alcuna logica che permetta di generare automaticamente test suite usando come punto di partenza la descrizione di un servizio web.

2.2.2 REST-Assured

Un'alternativa a Frisby.JS è rappresentata da REST-Assured. Si tratta di una libreria Java da utilizzare all'interno di JUnit. Il framework presenta caratteristiche molto simili a quelle presenti in Frisby.JS. Anche qui le chiamate sono concatenate grazie all'uso della funzione `then`. Più in generale, REST-Assured predilige lo stile `GivenWhenThen`. Questo schema segue tre idee principali:

- il metodo `given` descrive le precondizioni, ovvero lo stato delle cose prima che lo scenario descritto dal test si verifichi;
- la funzione `when` descrive l'evento da testare;
- il metodo `then` specifica quali sono i cambiamenti attesi nello scenario, ovvero le expectations.

Nel caso di REST-Assured con `given` si prepara la richiesta HTTP, specificando solitamente i parametri (path, query, body, ...), con il metodo `when` si effettua la chiamata, e, infine, la funzione `then` esegue le asserzioni sull'intestazione e il body della risposta.

Il framework fornisce anche alcune utility per maneggiare comodamente gli oggetti JSON. In particolare, è possibile validare il corpo JSON della risposta rispetto ad uno schema, la cui definizione è contenuta in un file esterno. REST-Assured introduce anche il supporto di JsonPath e XmlPath, attraverso i quali è possibile accedere a una serie di funzioni aggregate che rendono l'estrazione dei dati dalle risposte più semplice.

Le risposte ritornate dalle chiamate HTTP, sono automaticamente mappate su un oggetto appartenente ad una classe che espone dei metodi che danno accesso ai campi dell'header della risposta come ad esempio `getCookie` o `getStatusCode`.

Infine, c'è anche la possibilità di misurare il tempo di risposta, funzione tuttavia più utile nel caso di load testing che di functional testing.

In generale, REST-Assured si conferma un ottimo tool per il functional testing di servizi web, e costituisce un'ottima alternativa del mondo Java. Tuttavia, come per Frisby.JS, manca un interprete per descrizioni di servizi web.

2.2.3 PyRestTest

Prima di analizzare i framework in grado di generare codice automaticamente a partire da una descrizione del servizio web, è bene aprire una breve parentesi su un altro tipo di tool che trova in PyRestTest uno dei suoi migliori esponenti. In realtà questo tool, non è molto diverso da quelli visti finora. Ciò che cambia è l'approccio e lo stile con cui vengono descritti i test. I tool visti fino ad ora, infatti, si presentano come librerie da impiegare all'interno di programmi scritti in linguaggi di programmazione diversi, come Java o JavaScript. PyRestTest, invece, prevede la descrizione dei test in file YAML o JSON. Un'interprete Python si occupa poi di tradurre i test descritti.

Nella Fig. 2.2 viene mostrato un semplice test che, come nell'esempio fatto per Frisby.JS, invia una HTTP GET all'url `http://httpbin.org/status/418` e verifica che il codice di stato della risposta coincida con 418.

```
- test:
    - name: "Basic get"
    - url: "status/418"
    - method: "GET"
    - expected_status: [418]
```

Figura 2.2. Esempio di test descritto in YAML interpretabile da Phyresttest.

In questo caso, come si vede, il campo `url` del test è valorizzato con l'ultima parte dell'url completo, poichè la radice (`http://httpbin.org`) viene specificata nel momento in cui viene lanciato l'interprete. All'interno di ogni test è possibile specificare i campi dell'header, tramite il tag `header`, e il contenuto del body tramite il tag `body`. Anche qui, è possibile concatenare più test, e utilizzare, eventualmente, le risposte di un test come parametri per quelli successivi.

PyRestTest offre inoltre supporto per la validazione dei body JSON, grazie ai validatori. Esistono due tipi di validatori:

- il validatore dello schema JSON;
- il validatore dei valori contenuti nella risposta.

Per il validatore dello schema JSON, PyRestTest permette di specificare lo schema in un file separato. I validatori del secondo tipo, invece, utilizzano quelli che in PyRestTest vengono chiamati Extractors. Si tratta di funzioni in grado di estrarre, a vari livelli, valori dalla risposta. Esistono quattro tipi di extractor:

- jsonpath_mini: un estrattore base ispirato a JsonPath che usa notazione puntata e metodi di subscript/indexing (operatore [ ]) per estrarre i valori contenuti nel corpo della risposta;
- jmespath: fornisce un'implementazione completa di JMESPath, un linguaggio di query per JSON;
- header: estrae i valori dall'header della risposta HTTP;
- raw_body: un estrattore che dà accesso al corpo della risposta HTTP;

I validatori, oltre ad un extractor, ricevono, solitamente, come parametri in ingresso anche un comparator e il valore atteso. I comparator specificano la relazione di confronto tra il valore estratto e quello atteso. Il comparatore 'eq', per esempio, esprime la relazione di uguaglianza, mentre con 'contains' si richiede che il valore atteso sia contenuto nel valore estratto che sarà una collezione. Inoltre è possibile specificare come comparator anche una espressione regolare, tramite l'id 'regex'.

Nome generatore	*Tipo output*	*Descrizione*
number_sequence	Intero	Genera una sequenza di interi per cui è possibile specificare opzionalmente il punto di partenza e l'incremento
random_int	Intero	Genera un intero casuale
random_text	Stringa	Genera un stringa casuale per cui è possibile specificare opzionalmente la lunghezza minima e massima
choice	Qualsiasi	Seleziona un elemento di una lista

Tabella 2.1. Alcuni generatori disponibili in Phyresttest.

Infine, PyRestTest include al suo interno un insieme di generatori. Viene riportata una tabella (Tab. 2.1) che riassume le caratteristiche di alcuni dei generatori disponibili.

La presenza di questi moduli all'interno di PyRestTest costituisce un aspetto interessante soprtattutto se si pensa a quanto sia importante in un ambiente di testing disporre di dati da impiegare all'interno dei test. In questo caso, PyRestTest sceglie l'approccio più semplice, ovvero quello della generazione casuale dei dati.

Prima di chiudere la trattazione sul tool ed evidenziare gli svantaggi principali è bene aggiungere che PyRestTest include anche un modulo per il benchmarking. Questo modulo non è usato per il functional testing, bensì per i test di carico. Infatti non effettua validazioni sulle risposte HTTP, ma raccoglie statistiche dalle risposte. Di solito si effettuano un gran numero di richieste a un certo endpoint per misurare dati come per esempio tempo di risposta per ogni richiesta, tempo medio di risposta, velocità di download, velocità di upload.

PyRestTest si è dimostrato un tool interessante. I vantaggi principali sono sicuramente la presenza dei generatori, che consente di inviare richieste con dati sempre diversi, e la possibilità di descrivere i test in un semplice file di configurazione senza la necessità di conoscere uno specifico ambiente di programmazione ma solo le poche regole di YAML e JSON. La mancanza principale è, anche qui, il fatto di non poter generare i test a partire da una descrizione di un servizio, nonostante i test in PyRestTest sono molto simili a dei file di configurazione e non dei programmi scritti in un linguaggio di programmazione, cosa che dovrebbe rendere la generazione più agevole. Inoltre, PyRestTest presenta delle limitazioni sintattiche come ad esempio l'assenza di costrutti per supportare l'iterazione.

2.2.4 Swagger-test

Il primo tool che viene presentato in grado di generare test automaticamente a partire da una descrizione di un servizio è Swagger-test. Si tratta di un tool ancora lontano da quello che sarà il progetto finale, tuttavia propone un nuovo approccio al functional testing. L'idea di base è quella di testare un servizio scrivendo poche righe di codice JavaScript, quelle necessarie a inizializzare il framework. Il punto di partenza di Swagger-test, e dei tool appartenenti a questa seconda categoria, è una descrizione Swagger del servizio web che si desidera testare, di cui viene effettuato il parsing, in modo da disporre della descrizione come oggetto JSON all'interno del programma.

Swagger-test prevede due modi per generare i test. Il primo sfrutta un'estensione della specifica Swagger nota con il nome di `x-amples`. Questa estensione permette di specificare insiemi di coppie. Ogni coppia è costituita dalla richiesta, dalla corrispondente risposta attesa, ed opzionalmente include anche una descrizione testuale. Il tag `x-amples` può essere specificato per ogni combinazione di path e metodo. Questo primo tipo di generazione dei test richiede che la descrizione contenga tutte le informazioni che verranno usate dai test. Queste informazioni devono quindi essere inserite manualmente nella descrizione. Inoltre l'asserzione preferita dal tool si limita a confrontare, campo per campo, la risposta ottenuta con quella attesa, pertanto bisogna conoscere a priori quale sarà la risposta attesa. Il secondo metodo per generare i test conta sulla presenza di esempi contenuti all'interno della descrizione. Swagger consente di specificare, tramite il tag `example`, degli esempi per parametri e valori delle risposte. Anche in questo caso, però, questi valori vanno specificati

all'interno della descrizione. Tuttavia quest'ultimo tag è più comune rispetto al precedente, ma, essendo comunque opzionale ai fini della validità della descrizione, è facile trovare descrizioni che non lo impiegano.

Come visto Swagger-test fa leva sulla descrizione stessa per capire quali dati impiegare nei test, non solo per i parametri da usare nelle richieste che li prevedono, ma anche per la parte della validazione. Quello dei valori dei parametri infatti è un problema comune alla maggior parte dei tool di questo tipo.

Come detto in precedenza, Swagger-test non è un tool completo, infatti non genera codice compilabile, rieseguibile e modificabile e tutte le richieste e le asserzioni avvengono all'istante, ma rappresenta comunque un primo passo verso un ambiente a generazione automatica di test.

2.2.5 Dredd

Dredd è un tool language-agnostic usato per validare l'implementazione di un servizio web disponendo della descrizione. I linguaggi di descrizione supportati sono Swagger e API Blueprint. Si tratta di un programma scritto in CoffeScript, un linguaggio basato su JavaScript ma arricchito con alcuni costrutti tipici di Ruby e Python, eseguibile, come tutti i tool basati su JavaScript visti finora, con Node.js. Anche qui il punto di partenza è costituito dal file di descrizione dell'API, espressa in uno dei due linguaggi supportati. Come da documentazione, il funzionamento di Dredd si sviluppa in cinque fasi principali:

- Parsing e validazione sintattica della descrizione del servizio;
- Validazione semantica e creazione delle richieste e delle risposte attese;
- Invio delle richieste HTTP al servizio da testare;
- Verifica che le risposte coincidano con quelle attese;
- Resoconto dei risultati.

La fase più importante è certamente la seconda. Dredd, come Swagger-test, punta ad avere il contenuto delle risposte attese già nella descrizione. Per questo motivo, gli stessi autori nella documentazione suggeriscono agli utenti di modificare la loro descrizione del servizio che intendono testare prima di usare Dredd:

> It's very likely that your API description will not be testable **as is**. [10]

La documentazione quindi aiuta l'utente a modificare la propria descrizione in modo da renderla testabile tramite Dredd. Il primo aspetto riguarda i parametri che si specificano nell'URL. Solitamente nell'URL si specificano due tipi di parametri. I parametri del primo tipo appartengono al percorso, pertanto sono spesso battezzati parametri di tipo `path`. I parametri del secondo tipo, invece, sono quelli specificati alla fine di un URL nella sezione querystring introdotta dal simbolo "?". In questa sezione i parametri, noti con il nome di `query`, sono separati dal simbolo "&" e specificati secondo lo schema: `http://www.example.com?param1=value1&param2=value2`. Per ogni parametro di tipo query o path Dredd cerca i valori da usare nelle richieste sotto i tag `x-example`, un'estensione disponibile in Swagger, o `default`. L'assenza di uno di questi tag nella sezione di ogni parametro provocherà la terminazione dell'esecuzione e genererà un messaggio di errore che comunica l'ambiguità del parametro. Per quello che riguarda invece il body JSON delle richieste, la documentazione distingue tra descrizioni API Blueprint e Swagger. Per le descrizioni del primo tipo Dredd cerca gli esempi nella sezione `+Body` o `+Attribute`. Per Swagger invece gli esempi sono letti dalla sezione `schema.example` del parametro se presente, in alternativa Dredd genererà dei valori a partire dagli esempi contenuti nello schema del tipo che compare nella sezione delle dichiarazioni degli schemi stessi.

Dredd offre anche supporto per l'esecuzione di codice attorno a ogni test e da accesso agli oggetti contenenti le transazioni HTTP tramite gli Hook. Gli Hook si possono scrivere in tanti

linguaggi di programmazione come JavaScript, Perl, PHP, Python e Ruby. Alcuni esempi di Hook supportati sono beforeAll, afterEach e beforeValidation.

Come si vede Dredd è molto simile a Swagger-test a meno di qualche funzionalità aggiuntiva come quella degli Hook. In generale, però, il fatto che non generi del codice e il fatto che ricerchi i valori da assegnare ai parametri nella descrizione rappresentano due grandi svantaggi.

2.2.6 Altri tool

Prima di presentare l'ultimo tool conviene citare altri tool che sono stati provati ma che hanno caratteristiche molto simili a quelli già visti:

- Chakram: un tool molto simile a Frisby.JS;
- Webinject: un programma sviluppato tra il 2004 e il 2006, che usa lo stesso stile di PyRest-Test;
- JMeter: un programma di Apache scritto in Java e dotato di GUI più orientato al load testing che al functional testing;
- Swaggest-test: un tool basato su Swagger-test;
- SoapUI: un programma di SmartBear, dotato di GUI, in grado di generare automaticamente chiamate HTTP per un servizio a partire dalla sua descrizione Swagger.

Proprio SoapUI segue lo stesso stile di swagger-test-templates, il programma su cui si fonda il progetto finale.

2.3 Swagger-test-templates

Swagger-test-templates, che da qui in poi abbrevieremo con la sigla STT, rispetto ai programmi analizzati finora, presenta una caratteristica in più. STT infatti è un tool in grado di generare codice usando la descrizione Swagger dell'API. Il tool è stato sviluppato da un'azienda statunitense, Apigee, attiva nel campo dell'API management e acquisita da Google nel 2016. STT è disponibile come modulo di Node.js, su Github, e installabile tramite il comando `npm`. Si tratta quindi di un programma scritto JavaScript, privo di interfaccia grafica ed eseguibile col comando `node`.

STT si serve di una serie di librerie JavaScript, ad ognuna delle quali è affidato un compito preciso. Per fronteggiare il problema di generare codice JavaScript valido per i test, STT impiega Handlebars JS, una libreria che consente di creare template usando Mustache.JS, l'implementazione JavaScript del sistema di templating noto con il nome di Mustache[1].

```
source = fs.readFile(template.handlebars)
var template = Handlebars.compile(source);
var data = { "name": "Alan", "hometown": "Somewhere, TX", "kids": [{"name":
    "Jimmy", "age": "12"}, {"name": "Sally", "age": "4"}]};
var result = template(data);
```

Figura 2.3. Il programma JavaScript per generare il codice HTML usando Handlebars JS

[1]Il nome Mustache, in inglese baffi, deriva dall'uso intensivo nel sistema delle parentesi graffe, { }, che, ruotate, ricordano dei baffi (∽).

Per comprendere meglio di cosa si tratta, viene riportato un esempio presente nella pagina Github di Handlebar.JS (https://github.com/wycats/handlebars.js/#usage). La Fig. 2.4 mostra due codici HTML molto simili. Il codice a sinistra è il template. Il template HTML contiene delle stringhe racchiuse tra doppie parentesi graffe. Si tratta di placeholder o variabili che da sole non hanno alcun significato, ma che acquisiscono significato nel momento in cui viene specificato il contesto. Per fornire il contesto è sufficiente eseguire un programma come quello specificato nella Fig. 2.3.

```
<p>Hello, my name is {{name}}. I
    am from {{hometown}}. I have
    {{kids.length}} kids:</p>
<ul>
  {{#kids}}<li>{{name}} is
      {{age}}</li>{{/kids}}
</ul>
```

```
<p>Hello, my name is Alan. I am
    from Somewhere, TX. I have 2
    kids:</p>
<ul>
  <li>Jimmy is 12</li>
  <li>Sally is 4</li>
</ul>
```

Figura 2.4. A sinistra il template originale. A destra il risultato generato da Handlebars JS

Eseguendo il programma si ottiene il codice HTML che occupa la colonna di destra nella Fig. 2.4. Come si vede il programma JavaScript ha usato il contenuto della variabile `data`, un oggetto JSON, per "riempire" il template opportunamente. Le prime due sostituzioni, quelle di {{`name`}} e {{`hometown`}}, sono banali. Anche la successiva è facilmente intuibile. Qui infatti si usa la lunghezza dell'array `kids`. La sostituzione più interessante è sicuramente l'ultima, quella contenuta nel tag `<ul>`. Il carattere '#' posto davanti a `kids` ha lo stesso effetto di un ciclo for. In pratica l'espressione contenuta nella lista non ordinata si può tradurre come: per ogni kid contenuto nell'array kids, crea un nuovo elemento dell'elenco puntato con scritto "`kid.name` is `kid.age`". Il carattere '#' assume un altro significato se posto davanti a una variabile di tipo Booleana. In quel caso si ha lo stesso comportamento di un'istruzione condizionale. Per questi motivi, Mustache è definito un linguaggio "logic-less", ovvero privo di strutture di controllo esplicite (selezione e ciclo), e l'intero controllo del flusso dipende dai dati.

STT utilizza due diverse librerie per effettuare chiamate HTTP:

- Request;
- SuperTest.

All'utente viene data la possibilità di indicare quale libreria impiegare quando si lancia il programma. Per ognuna delle due librerie sono disponibili sette template. Ogni template è relativo a un metodo HTTP supportato. I metodi HTTP riconosciuti da STT nell'ottica della generazione dei test sono:

- GET;
- POST;
- PUT;
- DELETE;
- HEAD;
- OPTIONS;
- PATCH.

Per quello che riguarda le due librerie, la prima costituisce un client HTTP, la seconda, invece, oltre a implementare il protocollo HTTP, include, come si intuisce dal nome, degli strumenti per fare validazioni, e rappresenta quindi un'interfaccia ad alto livello per effettuare chiamate HTTP e validazioni sulle risposte.

Il framework scelto come ambiente di testing è Mocha JS. Mocha JS è un framework eseguibile con Node.js usato per test asincroni. Mette a disposizione una serie di alternative per descrivere test. Supporta test sincroni e asincroni e permette l'uso delle promise e degli hooks. STT sceglie l'approccio più semplice, ovvero quello del codice asincrono. Questo approccio prevede l'aggiunta di una callback, comunemente chiamata `done`. L'invocazione della callback stabilisce la fine del test, e accetta come parametro un oggetto della classe `Error`.

Per l'aspetto delle validazioni, STT si affida a Chai JS, un framework di testing che assume la forma di una libreria di validazione in stile BDD[2]/TDD[3]. Chai JS espone tre diverse interfacce per esprimere le validazioni:

- Assert;
- Expect;
- Should.

Di seguito verranno introdotti i tre stili di validazione, anche con degli esempi disponibili nella documentazione di Chai JS (http://chaijs.com/guide/styles/). Il primo stile è quello esposto dall'interfaccia `assert`. Per accedere ai metodi di validazione esposti dal modulo si usa la notazione puntata. I metodi accettano in ingresso almeno due parametri. Il primo è la variabile su cui effettuare la validazione. Il secondo rappresenta il valore atteso. Inoltre tutti i metodi permettono di aggiungere un terzo parametro opzionale, che è un messaggio da mostrare nel caso la validazione fallisca.

```
var assert = require('chai').assert
  , foo = 'bar'
  , beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };

assert.typeOf(foo, 'string'); // without optional message
assert.typeOf(foo, 'string', 'foo is a string'); // with optional message
assert.equal(foo, 'bar', 'foo equal bar');
assert.lengthOf(foo, 3, 'foo's value has a length of 3');
assert.lengthOf(beverages.tea, 3, 'beverages has 3 types of tea');
```

Figura 2.5. Esempio di script che usa la libreria `assert`.

Le restanti interfacce sono relative allo stile BDD. Entrambe prediligono uno stile di chiamate concatenabili che rende l'intera espressione molto simile a una frase del linguaggio umano. Nella Fig. 2.6 viene riportato uno script che impiega entrambe le interfacce. Come si vede l'interfaccia `expect` accetta un solo parametro che rappresenta l'oggetto su cui effettuare le validazioni. Lo stile `should` estende invece la classe a cui appartiene ogni oggetto aggiungendo la proprietà omonima che inizia la catena di validazione.

La differenza principale dei due stili, come si può notare dalle prime due righe dell'esempio, sta nel fatto che, mentre le validazioni in stile `expect` utilizzano il riferimento alla funzione `expect` del modulo di Chai, lo stile `should` richiede che la funzione `should` sia realmente eseguita.

[2]BDD: behavior-driven development

[3]TDD: test-driven development

```
var expect = require('chai').expect
var should = require('chai').should()
  , foo = 'bar'
  , beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(beverages).to.have.property('tea').with.lengthOf(3);
foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
beverages.should.have.property('tea').with.lengthOf(3);
```

Figura 2.6. Esempio di script che usa la libreria `expect` e `should`.

Inoltre, adottare lo stile `expect` risulta più sicuro dell'uso dello stile `should`. L'invocazione della funzione `should()` su una variabile che è `null` o `undefined` risulterebbe in un'istruzione non valida.

Dal punto di vista dell'uso pratico di STT, a disposizione del programmatore esiste un unico metodo globale chiamato `testGen` che ha il compito di generare i file dei test. Questo metodo accetta in ingresso due parametri:

- il file JSON che contiene la descrizione del servizio in OpenAPI/SWAGGER;
- un oggetto che funge da configuratore.

Attraverso l'oggetto di configurazione è possibile modificare il comportamento e il risultato finale dell'esecuzione della funzione. Gli argomenti settabili sono:

- `assertionFormat`: è un parametro obbligatorio che consente di scegliere lo stile di validazione che sarà seguito nei test. Le scelte possibili sono `should`, `expect` e `assert`;
- `testModule`: è un parametro obbligatorio che consente di scegliere con quale libreria effettuare le chiamate al servizio. Le scelte possibili sono `supertest` e `request`;
- `pathName`: è un parametro richiesto che accetta un array di path name, ovvero endpoint del servizio da testare, per il quale generare i test. Se la lista viene lasciata vuota STT genererà di default i test per tutti gli endpoint riportati nella descrizione;
- `statusCodes`: un parametro opzionale che accetta un array di codici di stato, espressi come interi. Se questo parametro viene valorizzato STT genererà i test solo per i codici di stato specificati, altrimenti saranno generati i test per tutti i codici di risposta riportati nella descrizione per ogni coppia path-metodo;
- `loadTest`: un parametro opzionale che permette di effettuare test di carico sul servizio. Il parametro accetta in ingresso un array di oggetti in cui si specifica l'endpoint del servizio, il metodo HTTP e un ulteriore oggetto `load` che contiene le informazioni sul numero di richieste totali da fare e il numero di richieste concorrenti. Quest'ultimo oggetto è a sua volta opzionale e se non specificato vengono usati i valori di default che sono rispettivamente 1000 e 100;
- `maxLen`: un parametro opzionale usato per troncare la lettura della descrizione. Di default vale 80. Se settato a -1 la descrizione viene letta interamente;

- `pathParams`: un parametro opzionale molto utile che permette di specificare i valori associati ai parametri di tipo path che appaiono negli endpoint riportati nel parametro `pathName`;
- `templatesPath`: un parametro opzionale di tipo stringa che permette di specificare il path di template handlebars personalizzati. Se specificato STT genera i test impiegando i nuovi template e non quelli di default;
- `requestData`: è un parametro opzionale estremamente utile che permette di specificare quali sono i valori dei parametri da usare nei test per ogni terna path-metodo-codice di risposta. Ovviamente questi parametri possono essere specificati solo per quelle coppie path-metodo che accettano parametri in ingresso.

La funzione `testGen` ritorna un array di oggetti. Ogni oggetto è formato da una coppia di stringhe. La prima stringa (proprietà `name`) è l'identificativo del file contenente un singolo test. Questo id è costruito usando il nome della descrizione del servizio e il nome del path a cui si riferisce il test. La seconda stringa (proprietà `test`) contiene invece il file stesso, ovvero il codice del test. A partire da queste informazioni ritornate da `testGen` è possibile salvare i file tramite `fs.writeFile(path, data, callback)` ed eseguirli successivamente.

Nella Fig. 2.7 viene mostrato un programma che usa STT.

```
var stt = require('swagger-test-templates');
var swagger = require('/path/to/swagger.json');
var config = {
  assertionFormat: 'should',
  testModule: 'supertest',
  pathName: ['/user', '/user/{id}'],
  maxLen: -1,
  pathParams: {
    "id": "123"
  }
};
var tests = stt.testGen(swagger, config);
var i = 0
for(; i< tests.length ; i++) {
  fs.writeFile(tests[i].name, tests[i].test, function(err) {
    if(err) {
      console.log(err);
      return
    }
  });
}
```

Figura 2.7. Esempio di programma che usa STT per generare i test.

Per capire meglio come funziona STT e che risultati produce si consideri un servizio di eco che espone un unico endpoint che accetta in ingresso un parametro query di tipo `string` e che risponde con un oggetto JSON contenente una stringa che rappresenta il messaggio di eco del server. Una volta letta la descrizione del servizio, e lanciato uno script simile a quello mostrato in Fig. 2.7, STT genererà un unico file, relativo all'unico endpoint esposto dal servizio, di cui una parte è riportata nella Fig. 2.8.

Le funzioni `describe()` alla riga 1 e 2 introducono un nuovo gruppo di unit test. In particolare, la prima è usata per raggruppare le unit test relative al path `greeting`, la seconda, invece, raggruppa per metodo HTTP. La funzione `it()` segna poi, alla riga successiva, l'inizio della prima unità di test. Come si può notare la singola unità di test si divide in due fasi.

```
1  describe('/greeting', function() {
2    describe('get', function() {
3      it('should respond with 200 Returns the greeting.', function(done) {
4        var schema = { "type": "object",
5                       "properties": {"message": {"type": "string"}},
6                       "required": ["message"]};
7        request({url: 'http://localhost:8080/greeting',json: true,
8          qs: { user: 'DATA GOES HERE'},
9          method: 'GET', headers: {'Content-Type': 'application/json'}
10       }, function(error, res, body) {
11           if (error) return done(error);
12           expect(res.statusCode).to.equal(200);
13           expect(validator.validate(body, schema)).to.be.true;
14           done();
15       });
16     });
17   });
18 });
```

Figura 2.8. Esempio di test generato per il servizio di eco.

Nella prima fase, dopo la dichiarazione dello schema JSON, che verrà usato successivamente, viene effettuata la chiamata HTTP. L'esempio qui utilizza la libreria `request`. La funzione `request` è invocata passando come parametro un oggetto contenente alcune informazioni necessarie a completare la richiesta HTTP. Tra queste informazioni ci sono, per esempio, l'url e il metodo HTTP. Alla riga 6, compare invece un'altra informazione importante: l'attributo `qs`. La proprietà `qs` sta per query string ed è quindi il canale per comunicare quali sono i valori che devono assumere i parametri di tipo query accettati dall'endpoint. Qui però il parametro `user` è valorizzato con la stringa `'DATA GOES HERE'`. Di default, infatti, STT non è in grado di creare i dati con cui confezionare le richieste HTTP, e il placeholder `'DATA GOES HERE'` indica la necessità di valorizzare manualmente i parametri di input delle richieste. Questo è il comportamento di default di STT. Ovviamente, questo si verifica anche per i parametri di tipo `path` e i `body` e non solo per i parametri di tipo `query`.

La seconda fase del test è invece eseguita in maniera asincrona, ovvero alla ricezione della risposta alla richiesta HTTP appena effettuata. Sostanzialmente in questa seconda fase viene asserito che:

- il codice della risposta coincida con uno di quelli specificati nella descrizione del servizio per la corrente coppia path-metodo HTTP;
- il corpo della risposta sia conforme alla schema dichiarato nella descrizione per la combinazione path-metodo-codice di stato della risposta.

Quest'ultima asserzione utilizza un oggetto validator esposto dalla libreria `ZSchema`. Ovviamente se la risposta non restituisce alcun dato la seconda asserzione cambia, e si verifica quindi che il corpo della risposta sia vuoto che si traduce nell'avere uno schema `null`. Nell'esempio viene usato lo stile `expect`, tuttavia è possibile ottenere le altre due versioni specificando lo stile desiderato tramite il parametro `assertionFormat` della funzione `testGen`, in fase di generazione dei test. Inoltre, cambiando lo stile di validazione le asserzioni rimangono le medesime.

Tornando ad analizzare la prima parte del test, se il test venisse eseguito così come è, la richiesta HTTP che verrebbe fatta sarebbe simile a quella mostrata nella Fig. 2.9. Come si vede il valore associato nel test al parametro query `user` viene usato proprio per effettuare la richiesta HTTP. In questo caso il servizio risponderebbe con `200 OK` e il body della risposta conterrebbe un oggetto JSON con un unica proprietà chiamata `message` (`{"message" : "Hello DATA GOES HERE!"}`)

```
GET /greeting?user=DATA+GOES+HERE HTTP/1.1
Host: localhost:8080
Accept: application/json
```

Figura 2.9. Richiesta HTTP semplificata eseguita dal test in figura Fig. 2.8 .

Ovviamente il fatto che il servizio in questo caso risponda con una risposta di successo è casuale. Il test infatti ha ancora il placeholder `'DATA GOES HERE'`, tuttavia essendo una stringa la richiesta che viene prodotta è una richiesta legittima per il servizio che risponde in maniera positiva.

In generale, infatti, l'idea di base in STT, nel suo funzionamento di default, è quella di produrre i test, ma la valorizzazione dei parametri spetta all'utente.

Tuttavia, come si era già anticipato nella sezione relativa ai parametri di input accettati dalla funzione `testGen`, esiste un parametro che permette all'utente di ottenere dei test che includono i dati, e per farlo è sufficiente preparare opportunamente un oggetto che ha una certa struttura e contiene i dati desiderati dall'utente e con questo valorizzare il parametro `requestData`. I dati usati in questa modalità prendono il nome di mock[4]. L'oggetto da passare al parametro `requestData` deve avere la struttura riportata nella figura Fig. 2.10.

```
{
   '/endpoint1': {
       operation1: {
           'responseCode1': [{ body: {}, description:' '}],
           'responseCode2': [{ queryParam1: 'value1', description:' '}]
       },
       operation2: {
           'responseCode3': [{ queryParam2: 'value1', description:' '},
                             { queryParam2: 'value2', description:' '}]
       }
   },
   '/endpoint2': {
       operation1: {
           'responseCode1': [{ pathParam1: 'value3', description:' '}]
       }
   }
 }
```

Figura 2.10. Struttura di `requestData`.

Attraverso questa struttura è possibile specificare per ogni endpoint esposto, per ogni metodo accettato dall'endpoint e per ogni codice di stato ritornato dal metodo, i valori che devono assumere i parametri. Nella struttura ad ogni terna è associato un array di oggetti. Lo schema di quest'ultimi varia a secondo del numero e dei tipi di parametri accettati, ed è unico per ogni terna. Tutti però includono un attributo, `description`, che ha lo scopo di rendere più verbosi i test. Il contenuto di `description` infatti viene aggiunto alla stringa passata in ingresso alla funzione `it()` che inizia il test, e quando questo viene eseguito il messaggio completo viene visualizzato sul terminale. Ogni oggetto presente nell'array di una singola terna provoca la creazione di un test. Quindi il numero

[4]mock: letteralmente *finto*, in questo contesto è usato per indicare dei dati inventati, simulati

di test generati per ogni terna è uguale alla dimensione dell'array associato alla terna stessa in `requestData`.

Tra tutti i tool analizzati fino ad ora STT risulta essere il più completo, promettente ed estendibile. Ciò che lo pone infatti davanti a quelli che in questa trattazione si possono considerare dei concorrenti è la capacità di produrre codice in maniera completamente automatica. Questo aspetto assume un'importanza fondamentale nell'ottica dell'automation testing. Il tool infatti permette concretamente di risparmiare il tempo necessario a preparare i test manualmente. Per quello che riguarda i prerequisiti, STT come si è visto necessita solo di una descrizione del servizio. Se non si dispone di una descrizione del genere, realizzarne una è un'operazione poco dispendiosa anche perché OpenAPI è un linguaggio estremamente intuitivo. Un altro aspetto interessante è la possibilità di modificare i risultati finali della generazione tramite la costruzione di template personalizzati, scritti seguendo le regole di Mustache.JS.

Bisogna tuttavia ricordare che STT è il risultato di poche settimane di lavoro del team di Apigee, pertanto è possibile riscontrare malfunzionamenti o comportamenti inaspettati. Tuttavia il contributo della community di Github fa sì che il tool sia continuamente aggiornato e migliorato in seguito a richieste di aggiunta di funzionalità o segnalazioni di bug. Uno dei difetti che si è riscontrato, per esempio, e che si è migliorato è l'impossibilità, fissata una terna path-metodo-codice di stato che accetta un parametro di tipo path, di generare più test assegnando valori diversi al parametro di tipo path. L'implementazione originale prevedeva che, fissata una terna che accetta in ingresso un parametro path, e definito l'array, contenente i valori da associare ai parametri di input, che compare in `requestData`, STT generasse per la terna un numero di test pari alla dimensione dell'array come atteso, tuttavia nei test generati il valore associato al parametro di tipo path era sempre quello che compariva nell'oggetto in testa all'array.

Inoltre, attualmente la logica di parsing contenuta nel programma è pensata per la versione 2 di OpenAPI. Pertanto il tool funziona solo con descrizioni che usano la versione 2 di OpenAPI. La versione 3 è stata pubblicata nel luglio del 2017. Tuttavia esistono diversi tool gratuiti che si occupano di tradurre descrizioni da una versione all'altra e addirittura da un linguaggio di descrizione a un altro, come ad esempio swagger2openapi di Mermade e api-spec-converter di LucyBot-Inc, entrambi disponibili su Github.

2.4 Il progetto finale

Finora sono stati presentati software, proprietari e open source, che offrono strumenti diversi per creare ed eseguire test funzionali di servizi web. Ciò che emerso dall'analisi fin qui fatta è che nell'ottica di rendere la creazione e l'esecuzione di test quanto più automatizzate possibile il framework migliore e più adatto è sicuramente l'ultimo ad essere stato presentato. L'unica caratteristica che manca a Swagger-test-templates, che lo renderebbe un tool completo, è la presenza di un modulo in grado di generare automaticamente i valori da assegnare ai parametri previsti dal servizio per ogni endpoint. Tra i tool visti in precedenza l'unico a includere un generatore molto semplice è PyRestTest.

2.4.1 I generatori

L'idea di base quindi, su cui si fonda la progettazione e la realizzazione del tool finale, è quella di sfruttare le capacità di Swagger-test-templates integrando un modulo preesistente in grado di fornire i valori da assegnare ai parametri, ovvero un generatore. Ovviamente questo generatore non può essere un generatore qualunque, ma deve comprendere i tipi e i formati dei dati che compaiono nelle descrizioni Swagger. Si è scelto quindi di impiegare un generatore casuale non eccessivamente complicato capace di produrre campioni di dati basandosi su gli schemi JSON presenti nella descrizione Swagger. Il generatore si chiama json-schema-test-data-generator ed è un modulo di bojand disponibile su Github (`https://github.com/bojand/json-schema-test-data-generator`). Il modulo è installabile attraverso il comando `npm install json-schema-test-data-generator`. Esporta un'unica funzione che accetta in ingresso uno schema JSON o un tipo primitivo tra quelli impiegabili nello schema, come `string`, `boolean` o `number`.

La funzione esposta dal modulo ritorna un array di oggetti, che seguono un formato preciso, e rappresentano delle combinazioni generate a partire dallo schema. Le combinazioni che vengono generate cercano di coprire vari casi per i test. Il formato degli oggetti generati viene riportato in Fig. 2.11.

```
{
  valid:   //boolean: indica se il dato è valido o no
  data:    //object: il valore generato
  message: //string: un messaggio che descrive il dato del test
  property://string|undefined: la proprietà dello schema a cui si riferisce la
      combinazione generata
}
```

Figura 2.11. Il formato degli oggetti prodotti dal generatore.

Per comprendere meglio cosa si intende quando si parla di combinazioni e di casi di test si riporta in Fig. 2.12 un semplice script che, a partire da uno schema JSON, stampa su schermo le combinazioni generate.

```
var generate = require('json-schema-test-data-generator');

var schema = {
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "minLength:": 5
    },
    "active": {
      "type": "boolean"
    },
    "email": {
      "type": "string",
      "format": "email"
    },
    "accountNumber": {
      "type": "number"
    }
  },
  "required": ["name", "email"]
}

console.dir(generate(schema));
```

Figura 2.12. Uno script che genera i campioni per i test a partire da uno schema JSON.

Nello script viene dichiarato un semplice schema JSON con quattro proprietà (`name`, `email`, `active` e `accountNumber`). Per alcune di queste vengono specificate delle informazioni aggiuntive sul formato, come per esempio il vincolo che la proprietà `name` sia una stringa con almeno cinque caratteri. Lo schema specifica anche quali sono le proprietà obbligatorie, ovvero `name` e `email`. L'output dello script in Fig. 2.12 è riportato in Fig. 2.13. Tutti gli esempi qui riportati sono quelli

che compaiono nel file di documentazione di json-schema-test-data-generator, presente nella pagina Github del progetto.

```
1  [ { valid: true,
2      data: { name: 'Lorem', email: 'Rl5W1lWNWCxI2@IKXPOiX.tx' },
3      message: 'should work with all required properties' },
4    { valid: true, property: 'active',
5      data: {    name: 'in Excepteur',
6                 email: 'dI4QX7i3o4aW1@pCYSSCbdzKxinpBylf.bid',
7                 accountNumber: -45141884.82426107 },
8      message: 'should work without optional property: active'},
9    { valid: true, property: 'accountNumber',
10     data: {    name: 'consectetur amet dolor',
11                email: 'ylDMsjokBq6o@kyISBuRrISJSVGfps.chh' },
12     message: 'should work without optional property: accountNumber'},
13   { valid: false, property: 'name',
14     data: { email: 'inL@FijDPFt.jsv' },
15     message: 'should not work without required property: name'},
16   { valid: false, property: 'email',
17     data: { name: 'labore', active: true },
18     message: 'should not work without required property: email'},
19   { valid: false, property: 'name',
20     data: { name: true, email: '3JbKSBUkulRThv@BB.bbsf' },
21     message: 'should not work with \'name\' of type \'boolean\''},
22   { valid: false, property: 'name',
23     data: {    name: '4x%u',
24                email: '5j6OszCCU-gZFx4@htyNxrrFVtVJghtWRaXQhjALzJqisumh.pmz' },
25     message: 'should not pass validation for minLength of property: name'},
26   { valid: false, property: 'active'
27     data: {    name: 'deserunt nostrud dolore ea',
28                email: 'b9XLfGB3Bs@golsytCcKabldW.ufin',
29                active: 5313786068074496 },
30     message: 'should not work with \'active\' of type \'integer\''},
31   { valid: false, property: 'email',
32     data: {    name: 'cillum ',
33                email: null,
34                accountNumber: 67113212.30977774 },
35     message: 'should not work with \'email\' of type \'null\''},
36   { valid: false, property: 'email',
37     data: { name: 'anim laborum quis occaecat', email: '8WruHF' },
38     message: 'should not pass validation for format of property: email'},
39   { valid: false, property: 'accountNumber'
40     data: {    name: 'veniam nulla ut',
41                email: 'wYcn@RVogRzpzOxnmGQvStZmpMkil.czip',
42                accountNumber: 'snOS2H9j)]' },
43     message: 'should not work with \'accountNumber\' of type \'string\''} ]
```

Figura 2.13. L'output dello script di Fig. 2.12.

L'array generato dallo script ha undici elementi. Di questi, tre sono validi, mentre i restanti sette sono marcati come non validi dal generatore. Per ogni oggetto generato, il campo `message` e `property` forniscono informazioni aggiuntive circa la validità, rispetto allo schema, del campione generato.

Il primo oggetto proposto dal generatore è perfettamente conforme allo schema. Ha le sole

proprietà obbligatorie (`name` e `email`) valorizzate con dati validi. In particolare, il valore associato a `name` rispetta il vincolo di lunghezza minima, mentre il valore dato a `email` rispetta il formato di un indirizzo email. In questo caso il campo `property` non è definito e la descrizione specifica che l'oggetto presenta soltanto le proprietà che nello schema sono marcate come `required`.

Nel secondo oggetto generato, anch'esso valido, oltre alle proprietà obbligatorie, è valorizzato anche il campo opzionale `accountNumber`. Il messaggio indica che il valore generato è valido anche se la proprietà `active` non è definita. L'ultimo oggetto valido generato è simile al precedente ma `active` e `accountNumber` qui si scambiano i ruoli.

Il primo oggetto non valido inizia alla riga 13. Il generatore, tramite la proprietà `valid` e `message` segnala che l'oggetto non può essere considerato valido poiché la proprietà obbligatoria `name` non è definita. In questo caso, quindi, l'oggetto non è conforme allo schema JSON. Anche l'oggetto successivo dell'array non è valido. Qui, analogamente a quanto avviene nel caso precedente, manca la proprietà `email`.

Gli oggetti che iniziano alle righe 19, 26, 31 e 39 sono non validi e sono generati tenendo a mente la stessa regola. Il generatore, per ogni proprietà prevista nello schema JSON, propone un oggetto in cui alla proprietà presa in considerazione viene assegnato un valore di un tipo errato. Come si vede dalla Tab. 2.2 il generatore propone un tipo sbagliato per ogni proprietà specificata nello schema, sia per quelle obbligatorie che per quelle opzionali.

Proprietà JSON	*Tipo*	*Tipo errato*
name	String	Boolean
active	Boolean	Integer
email	String	null
accountNumber	Number	String

Tabella 2.2. I tipi errati proposti dal generatore per le proprietà JSON.

Vi sono poi altri due oggetti non validi proposti dal generatore. Entrambi sono marcati come non validi in quanto non rispettano i vincoli specificati dal formato o dalle informazioni aggiuntive contenute nello schema. In particolare, l'oggetto che inizia alla riga 22, risulta essere non valido poiché la proprietà `name` è valorizzata con una stringa di quattro caratteri. Un oggetto così fatto non può considerarsi valido rispetto allo schema in quanto il campo `name` deve essere una stringa di almeno cinque caratteri. In questo caso si può vedere che il generatore attribuisce al campo `property` il valore `name` e comunica tramite `message` il motivo della non validità dell'oggetto generato. L'altro oggetto non valido, individuato alla riga 36, riguarda invece il campo `email`. Lo schema si serve della proprietà `format` per specificare che il campo `email` non può essere una stringa qualunque e che deve rispettare la sintassi di un indirizzo email. Il valore di `email` (`8WruHF`) non rispetta chiaramente la sintassi richiesta e per questo motivo il generatore marca l'oggetto proposto come non valido e motiva tramite il messaggio la non validità dell'oggetto in questione (`should not pass validation for format of property: email`).

Json-schema-test-data-generator non è l'unico generatore che è stato valutato al fine di essere integrato nel progetto finale. Un altro generatore che è stato provato è swagmock. Swagmock è un generatore di dati creato apposta per Swagger, come si intuisce dal nome. Si tratta di un modulo scritto in JavaScript disponibile su Github come repository di subeeshcbabu (`https://github.com/subeeshcbabu/swagmock`). Come gli altri moduli anche Swagmock si installa tramite il comando `npm`.

Il modulo espone un'unica interfaccia chiamata proprio `Swagmock`. La funzione accetta due metodi:

- api: è il parametro (obbligatorio) che contiene l'informazione relativa alla descrizione Swagger. Può essere una delle seguenti cose:
  - il path relativo o assoluto che individua il file della descrizione;
  - l'url della descrizione;

  - un oggetto che costituisce la descrizione;
  - una promise che ritorna l'oggetto contenente la descrizione.

- options: è un oggetto opzionale contenente la proprietà `validated`, un booleano che se settato a `true` indica che la descrizione è stata validata e che tutti i $ref sono risolti.

Il modulo espone poi tre diversi metodi:

- responses;
- parameters
- requests.

Chiamando le funzioni sopra elencate si ottengono rispettivamente i body delle risposte, i parametri delle richieste e le richieste complete. Ovviamente si tratta di dati "mockati", finti ma verosimili, ovvero dati che potrebbero essere restituiti o accettati dal vero servizio che implementa la descrizione fornita in ingresso a Swagmock. Di nuovo, i dati in questione, le risposte per esempio, non sono in alcun modo ottenute contattando un ipotetico server che implementa il servizio della descrizione, ma sono generate offline, disponendo solo ed esclusivamente della descrizione del servizio.

Tutti i tre i metodi accettano in ingresso due parametri:

- options;
- callback;

Il primo parametro è un oggetto che ha proprietà diverse per ogni metodo esposto. La Tab. 2.3 sintetizza per ogni metodo quale proprietà di `options` ha senso valorizzare.

	path	operation	response	useExamples
responses	x	x	x	x
parameters	x	x		
requests	x	x		

Tabella 2.3. Le proprietà valorizzabili del parametro `options` per i tre metodi (la x indica che il campo si può valorizzare).

Le quattro proprietà dell'oggetto `options` istruiscono il generatore nella creazione dei mock. In particolare:

- `path` permette di specificare il path per cui generare il mock. Se non è specificato i mock sono generati per tutti i path della descrizione;
- `operation` permette di specificare il metodo HTTP per cui generare il mock. Se non è specificato sono generati per tutti i metodi del path;
- `response` permette di specificare per quale codice di stato il mock deve essere generato. Se non è specificato il mock è generato per tutte le risposte di una coppia path-metodo;
- `useExamples` un Booleano con cui si richiede al generatore di cercare nella descrizione i valori con cui generare i mock. Se `true` il generatore non genera i dati autonomamente ma usa quelli che nella descrizione compaiono sotto i tag `example` o `default`.

Il parametro `callback` è, come dice la parola stessa, una funzione che viene eseguita subito dopo che il metodo ha terminato la sua esecuzione, da cui riceve in ingresso un oggetto contenente il risultato, il mock, e un errore che indica l'esito dell'operazione. Il contenuto del mock varia in base a quale dei tre metodi esposti viene invocato.

Il metodo `response` genera come detto esempi di body delle risposte pertanto è più adatto in quelle situazioni in cui bisogna sviluppare programmi che fanno uso dei dati ricevuti da un servizio, ma il servizio non è ancora disponibile. In generale, questo tipo di mock aiuta gli sviluppatori front-end quando le risposte del back-end non sono ancora completamente disponibili.

I metodi `parameters` e `request` sono molto simili. Entrambi producono gli stessi dati ma in forma diversa, e generano oggetti con proprietà come `query`, `header`, `pathname`, `path`, `formData`, `body` in base ai tipi dei parametri specificati nella descrizione per ogni coppia path-metodo. In Fig. 2.14 viene riportato un esempio di mock generato da swagmock con il metodo `parameters`. Nella creazione del mock è stato specificato un singolo URL ma nessun metodo. L'URL in questione espone i soli metodi `get` e `delete`. Il path inoltre prevede un unico parametro di tipo `path` chiamato `orderId`. Per ogni parametro viene quindi specificato il valore generato sotto la proprietà `value`.

```
{
  "get": {
    "parameters": {
      "path": [{ "name": "orderId", "value": 9 }]
    }
  },
  "delete": {
    "parameters": {
      "path": [{ "name": "orderId", "value": 8573207911071745 }]
    }
  }
}
```

Figura 2.14. Un esempio di mock prodotto con `parameters`.

La struttura dell'oggetto generato segue quindi un preciso schema che può essere così riassunto: il mock contiene come proprietà gli endpoint, sotto ogni endpoint compare un oggetto che ha come proprietà i metodi previsti dall'endpoint, ad ogni metodo è associata la sola proprietà `parameters`, sotto alla quale possono comparire una o più delle proprietà riportate più sopra (`query`, `path`, `body` ecc.). Sotto ogni proprietà, infine, compare un array di oggetti aventi le proprietà `name` e `value`. L'array contiene ovviamente un oggetto per ogni parametro del tipo specificato.

Il motivo per cui json-schema-test-data-generator è stato preferito rispetto a swagmock a questo punto risulta abbastanza chiaro. Json-schema-test-data-generator è nativamente più orientato al testing. Swagmock invece è sì un generatore ma è utile solo per il mocking. Infatti swagmock è più utile per generare le possibili risposte del servizio che per generare le possibili richieste. La differenza principale tra i due generatori sta nel fatto che, per ogni parametro, mentre json-schema-test-data-generator fornisce più dati, validi e non validi, swagmock si limita a fornire un unico dato valido.

2.4.2 L'architettura dell'applicazione

Per l'applicazione si è scelto di partire da un modulo preesistente. Si tratta di un modulo chiamato openapi-test-generator (https://github.com/Remco75/openapi-test-generator). Il modulo rappresenta una primissima versione di quello che sarebbe dovuto essere un progetto più grande e completo ma l'ultimo commit risale all'ottobre 2017. Openapi-test-generator però si basa sulla

stessa idea esposta sopra, ovvero quella di integrare le capacità di swagger-test-templates con un generatore di dati per le richieste. In più il modulo prevede la creazione di mock per le risposte. Di fatto openapi-test-generator utilizza i due generatori presentati sopra. Riassumendo quindi, openapi-test-generator è in grado di:

- generare test usando l'accoppiata swagger-test-templates json-schema-test-data-generator;
- generare mock usando swagmock;

La Fig. 2.15 mostra uno schema semplificato con i principali componenti dell'applicazione.

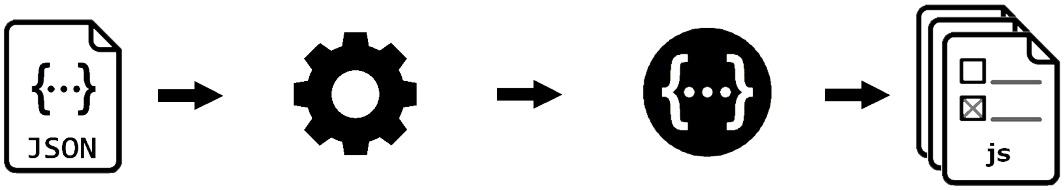

Figura 2.15. L'architettura dell'applicazione.

Il punto di partenza, come più volte detto, è la descrizione del servizio espressa nel linguaggio di descrizione Swagger/OpenApi. La presenza della descrizione risulta indispensabile, poiché al fine di testare un servizio web in maniera automatica bisogna conoscere in anticipo ciò che espone, la rappresentazione dei dati che scambia e, infine, quali sono le possibili risposte che può ritornare. In questo senso, la descrizione è in grado di soddisfare tutte le necessità indicate. Nello schema la descrizione è rappresentata dall'icona che raffigura il file JSON. Lo step successivo consiste nella creazione dei dati che compariranno come parametri nei test. Questo step coinvolge ovviamente il generatore json-schema-test-data-generator, rappresentato nella figura dall'ingranaggio. Il generatore utilizza la descrizione del servizio per generare le collezioni di dati da fornire in ingresso successivamente a swagger-test-templates, che in figura è invece illustrato con il logo di Swagger (tre punti mediani all'interno di due parentesi graffe). L'ultima icona della Fig. 2.15 rappresenta, infine, i test generati da swagger-test-templates. Ogni test è relativo ad un endpoint del servizio, e include diverse unità di test.

L'intero processo è coordinato dall'applicazione che interviene soprattutto nella parte centrale. Il suo compito infatti è, avendo a disposizione i dati prodotti dal generatore, organizzarli in un oggetto che segue la struttura desiderata da swagger-test-templates per la lettura dei dati per le richieste, ovvero l'oggetto da passare al parametro `requestData`.

L'applicazione è formata da più sorgenti, alcuni fanno parte del modulo openapi-test-generator, gli altri sono relativi a swagger-test-templates e json-schema-test-data-generator. Le modifiche su questi ultimi due sono state limitate ma comunque importanti al fine di ottenere un comportamento diverso rispetto a quello originale dei rispettivi moduli, tuttavia gran parte del lavoro si è concentrato sul codice di openapi-test-generator. Nel corso del capitolo verranno riportati i passaggi più importanti e le modifiche ai moduli esterni in modo da avere una visione completa del funzionamento dell'applicazione.

2.4.3 Il manuale del programmatore

Lo script che lancia l'applicazione è riportato nella Fig. 2.16. Lo script utilizza il modulo openapi-test-generator. Il modulo espone un costruttore che prevede in ingresso il file contenente la descrizione Swagger e una directory dove creare i test. Queste due informazioni devono essere fornite dall'utente da linea di comando quando lo script viene lanciato. I dettagli relativi a come fornire i parametri in input verrano approfonditi nella sezione relativa al manuale utente.

Una volta allocato l'oggetto si invoca il metodo `generate`. Il metodo accetta due parametri:

- writeMocks: un booleano che indica al generatore se deve o meno salvare i dati usati per fare le richieste. Di default è `false`;
- templatesPath: una stringa che indica la directory in cui si trovano i templates che dovranno essere usati per generare i test.

```
var OpenApiGenerator = require('open-api-test-generator');
var spec_name = process.argv[2];
var testDir = process.argv[3];
var pathParamValueDir = process.argv[4]
var spec = require(spec_name);
if (spec) {
        var generator = OpenApiGenerator(spec, testDir, pathParamValueDir);
        generator.generate(true);
}
```

Figura 2.16. Lo script principale che lancia l'applicazione.

Nello script il metodo `generate` è invocato ignorando il secondo parametro, dato che verranno impiegati i template di default previsti da swagger-test-templates, e settando a `true` il primo. In questo modo dopo la creazione dei test sarà possibile visualizzare quali dati sono stati generati per i test.

Il costruttore `OpenApiGenerator` effettua una serie di operazioni preliminari. Per prima cosa verifica che è stato correttamente invocato, cioè che entrambi i parametri siano stati forniti. Successivamente procede con la validazione della specifica Swagger. Per farlo utilizza la libreria ZSchema (https://github.com/zaggino/z-schema). Attraverso il metodo `validate` viene eseguita la validazione della descrizione fornita in ingresso rispetto allo schema JSON ufficiale di Swagger. Se anche soltanto uno dei controlli precedenti descritti fallisce viene lanciata un'eccezione specifica con un messaggio che spiega per quale motivo l'eccezione è stata lanciata. Viceversa, se tutti i controlli vanno a buon fine il programma prosegue la sua esecuzione effettuando un'operazione di dereferenziamento sulla descrizione, usando il metodo `deref` messo a disposizione dalla libreria json-schema-deref-sync(https://github.com/bojand/json-schema-deref-sync). L'obiettivo di questa operazione è quello di dereferenziare i puntatori JSON presenti nella descrizione, ovvero sostituire i riferimenti con i veri valori. In Fig. 2.17 viene mostrato il risultato che si ottiene chiamando il metodo `deref`. La colonna di sinistra mostra uno schema JSON che usa più volte la notazione di referenza ($ref) per puntare a definizioni che si trovano all'interno del documento, in rete o in un altro file del file system. La colonna di destra mostra invece come si presenta lo schema dopo aver invocato il metodo `deref`. Come si può notare, il primo uso di referenza viene sostituito con la definizione puntata che si trova alla riga 6. La libreria è in grado di dereferenziare non solo riferimenti semplici, ovvero puntamenti all'interno della stessa definizione, come nel caso precedente, ma anche più complessi come negli altri due casi (righe 18 e 22) in cui i riferimenti puntano a definizioni reperibili a un certo URL oppure contenute in un file diverso. L'uso dei riferimenti risulta estremamente utile e rende all'occhio umano le strutture JSON più leggibili, soprattutto se sono molto complicate. Inoltre, consente di evitare di dover riscrivere più volte la stessa definizione. Tuttavia, in questo caso, per il programma è più conveniente appiattire la descrizione ed esplicitare ogni riferimento.

Di ritorno dalla `deref` il prossimo passo consiste nel creare la struttura di directory che ospiterà i file contenenti i dati usati per ogni singola richiesta. In questa fase non risulta indispensabile conoscere la struttura nel dettaglio. Lo schema che è stato usato prevede di avere a livello più esterno le directory relative ai metodi HTTP. Dentro ognuna di queste si trovano delle directory annidate relative agli endpoint che prevedono il corrispondente metodo HTTP. I nomi delle directory annidate sono ricavate a partire dal nome dell'endpoint. In particolare le directory hanno gli stessi nomi delle risorse del servizio.

```
{
  "description": "Just some
      JSON schema.",
  "title": "Basic Widget",
  "type": "object",
  "definitions": {
    "id": {
      "description": "unique
          identifier",
      "type": "string",
      "minLength": 1,
      "readOnly": true
    }
  },
  "properties": {
    "id": {
      "$ref":
          "#/definitions/id"
    },
    "foo": {
      "$ref":
          "http://www.example.com/
          myschema.json#/definitions/foo"
    },
    "bar": {
      "$ref": "bar.json"
    }
  }
}
```

```
1  {
2    "description": "Just some
         JSON schema.",
3    "title": "Basic Widget",
4    "type": "object",
5    "definitions": {
6      "id": {
7        "description": "unique
             identifier",
8        "type": "string",
9        "minLength": 1,
10       "readOnly": true
11     }
12   },
13   "properties": {
14     "id": {
15       "description": "unique
             identifier",
16       "type": "string",
17       "minLength": 1,
18       "readOnly": true
19     },
20     "foo": {
21       "description": "foo
             property",
22       "readOnly": true,
23       "type": "number"
24     },
25     "bar": {
26       "description": "bar
             property",
27       "type": "boolean"
28     }
29   }
30 }
```

Figura 2.17. Il confronto tra lo stesso schema JSON prima (a sinistra) e dopo (a destra) il processo di dereferenziamento.

Successivamente il programma passa all'analisi della descrizione e alla generazione dei dati da usare per valorizzare i parametri per le coppie endpoint-metodo HTTP che li prevedono. Osservando lo schema di Fig. 2.15 questa fase coincide con il primo step. Per invocare il generatore, e preparare i dati, si passa dal metodo `generate`. Questo metodo si occupa principalmente di inizializzare correttamente l'oggetto `requestMocks`, ovvero l'oggetto che sarà poi dato in ingresso a swagger-test-templates come valore del parametro `requestData`. Come si vede dalla Fig. 2.18, la funzione `generate` cicla sugli endpoint della descrizione, e, per ognuno di essi, invoca `requestMockGenerator`. Non si tratta del vero generatore, bensì di un wrapper che invocherà il vero generatore e riempirà opportunamente `requestMocks`. Prima di invocare però il wrapper ci si accerta che le proprietà relative ad ogni endpoint della descrizione esistano. Se così non fosse il wrapper proverebbe a scrivere su una proprietà che risulta `undefined`. Al termine del ciclo sugli endpoint, avendo a disposizione l'oggetto `requestMocks` composto ad ogni ciclo dal risultato delle chiamate al wrapper, si invocherà `generateTest` per, come si può intuire dal nome, generare i test con swagger-test-templates.

```
function generate(writeMocks, templatesPath) {
  var requestMocks = {}
  var basePath = spec.basePath || '';

  flatPaths.forEach(function(apiPath) {
    if(!requestMocks[apiPath.path]) {
      requestMocks[apiPath.path] = {};
    }
    requestMocks[apiPath.path][apiPath.operation] =
        requestMockGenerator(apiPath.path, apiPath.operation, writeMocks);
  });
  generateTests(requestMocks, templatesPath, codes);
}
```

Figura 2.18. La funzione `generate`.

La prima cosa che viene verificata all'ingresso del wrapper è che la combinazione corrente di endpoint e metodo HTTP preveda almeno un parametro. Questa informazione ovviamente è ricavata leggendo la descrizione del servizio. Se la coppia non prevede parametri il generatore non viene impiegato. Se invece per la coppia sono previsti uno o più parametri si verifica la posizione del parametro, ovvero il valore della proprietà `in`. Il generatore viene invocato solo se il parametro non è un parametro di tipo `path`, cioè non è un parametro che individua una risorsa del servizio. Il motivo per cui si decide di filtrare i parametri di tipo path è legato alla difficoltà di generare dei dati validi in maniera casuale e si cercherà di spiegarlo con degli esempi.

Supponiamo, ad esempio, di voler testare un semplice servizio che permette di gestire un database di prodotti. Si immagini che il servizio esponga un endpoint, `/products`, a cui si può inviare una richiesta di tipo GET che restituisce una collezione di oggetti, ognuno dei quali rappresenta il singolo prodotto. In questo esempio conoscere lo schema di ogni oggetto prodotto non risulta indispensabile. È inoltre plausibile che il servizio esponga anche un altro endpoint, `/products/productId`, che restituisce la rappresentazione del prodotto che ha per identificativo proprio `productId`. In uno scenario del genere, un generatore casuale non può per sua natura proporre dei valori da associare al parametro `productId` che generino 200, cioè che abbiano una risorsa associata all'interno del servizio. È bene ricordare, infatti, che il generatore basa il suo funzionamento sulle informazioni contenute nella descrizione alla sezione dedicata ai parametri e alle definizioni, ovvero sul tipo, sul formato e su quei vincoli aggiuntivi come la lunghezza minima e/o massima per una stringa e il valore minimo e/o massimo per un numero. Inoltre, è bene aggiungere che l'applicazione utilizza il valore del campo `valid` degli oggetti ritornati dal generatore come discriminante per stabilire se il singolo oggetto proposto dal generatore debba essere considerato un parametro che provoca un success (codici di stato del tipo 2xx) o un client error (codici di stato del tipo 4xx). Il rischio che si correrebbe se si impiegasse il generatore per creare i valori anche per i parametri di tipo `path` è che si avrebbero dei test con l'asserzione di un success (expect(res.statusCode).to.equal(200)) in cui però al parametro `path` viene assegnato un valore che per il generatore è valido, in quanto rispetta i vincoli di tipo e formato, ma per cui non esiste una rappresentazione sul servizio.

Riprendendo l'esempio precedente, supponiamo che il parametro `productId` sia un intero compreso tra 1 e 10 estremi inclusi. Supponiamo inoltre che il database del servizio contenga solo 4 prodotti con id 1, 2, 6 e 10. La Tab. 2.4 aiuta a visualizzare come il servizio risponde inviando una GET all'endpoint `/products/productId` valorizzando il parametro dell'id con i valori accettati dal servizio, ovvero gli interi compresi tra 1 e 10 estremi inclusi. In base a quanto detto le uniche chiamate a restituire success (200) sono quelle in cui il valore di `productId` è 1, 2, 6 e 10. Per gli altri valori il servizio risponderà con client error (404).

Supponiamo di lanciare il generatore, fornendogli in input la descrizione del servizio che gestisce

ProductId	*Richiesta HTTP*	*Codice di stato della risposta*
1	GET /products/1	200
2	GET /products/2	200
3	GET /products/3	404
4	GET /products/4	404
5	GET /products/5	404
6	GET /products/6	200
7	GET /products/7	404
8	GET /products/8	404
9	GET /products/9	404
10	GET /products/10	200

Tabella 2.4. I codici di stato restituiti dalle richieste GET all'endpoint `/products/productId` per i valori compresi nell'intervallo di valori che rispettano i vincoli specificati nella descrizione.

il database di prodotti. E supponiamo inoltre che il generatore proponga tre oggetti, di cui uno marcato come valido, e i restanti due marcati ovviamente come non validi. Gli oggetti proposti potrebbero essere per esempio:

- ```
  { valid: true,
    data: 5,
    message: 'should work with all required properties' }
  ```

- ```
  { valid: false,
    data: 0,
    message: 'should not pass validation for minimum' }
  ```

- ```
  { valid: false,
    data: 11,
    message: 'should not pass validation for maximum' }
  ```

Il primo oggetto proposto dal generatore risulta valido in quanto il valore generato rispetta il tipo e il formato specificato per il parametro nella descrizione. Inoltre anche il vincolo relativo all'intervallo accettato è rispettato. Gli altri due oggetti generati, invece, non possono essere considerati validi in quanto, sebbene i dati associati rispettino i vincoli di tipo e formato, la condizione relativa all'intervallo di appartenenza è violata, infatti 0 e 11 non sono compresi nell'intervallo [1;10]. Si noti come il generatore, fissato l'intervallo, si limiti a individuare i suoi estremi a cui aggiunge o sottrae una unità per generare un valore non valido. In particolare, in caso di vincolo sul minimo, sottrae. Viceversa nel caso di vincolo sul massimo aggiunge. Ovviamente, non è detto che l'intervallo ammesso sia chiuso. Cioè può accadere che per un parametro si definisca nella descrizione solo il minimo o il massimo. Se per esempio, per il parametro `productId` la descrizione dichiarava il solo vincolo sul minimo (`productId` maggiore di zero), il generatore avrebbe proposto come valore non valido soltanto 0, seguendo la logica appena spiegata.

Test	*ProductId*	*Valido*	*Richiesta HTTP*	*Codice di stato*		*Esito test*
				Reale	Atteso	
A	5	Sì	GET /products/5	404	200	Fallito
B	0	No	GET /products/0	404	404	Superato
C	11	No	GET /products/11	404	404	Superato

Tabella 2.5. I codici di stato attesi per le richieste GET all'endpoint `/products/productId` per i valori proposti dal generatore.

Utilizzando questi tre oggetti e basandosi sul valore del campo `valid`, l'applicazione finirebbe con l'associare il primo e unico valore valido generato al codice di risposta 200, e i restanti valori al
  ```

codice di risposta 404. In questo modo si genererebbero tre test, che per comodità identificheremo con le lettere A,B e C. La Tab. 2.5 cerca di riassumere alcune informazioni importanti riguardo ai test generati. Per ogni test viene riportato nella tabella il valore scelto dal generatore da assegnare al parametro `productId` nella richiesta HTTP del test e se è valido o meno. Inoltre si specifica quali sono i codici di stato attesi su cui avviene l'asserzione e quello reale, e infine se il test, una volta eseguito nelle condizioni del servizio descritte nella Tab. 2.4, fallirà o sarà superato.

I test B e C vanno a buon fine. Il test A invece fallisce perché il codice di stato restituito dalla GET /products/5 è 404, come illustrato nella Tab. 2.4, ma l'applicazione ha generato il test con l'asserzione:

```
expect(res.statusCode).to.equal(200);
```

Il motivo per il quale un qualsiasi test dovrebbe fallire non può essere legato alle logiche di generazione del test stesso. In altre parole, il fallimento di un test dovrebbe essere dovuto esclusivamente a errori di implementazione del servizio o al massimo a discordanze tra l'implementazione del servizio e la sua descrizione. In questo caso però il test fallisce poiché l'applicazione ha generato un test con un'asserzione non corretta. Il motivo di questo errore è evidentemente la scelta di usare il generatore per decidere i valori da assegnare ai parametri di tipo `path`.

A questo punto resta da spiegare come è stato risolto il problema e quali soluzioni sono state prese in considerazione. Le soluzioni possibili sono due. La prima soluzione è la più semplice a cui si può pensare ed è anche quella che è stata alla fine adottata. Questa soluzione prevede che sia l'utente a dover indicare quali sono per ogni terna endpoint-metodo HTTP-codice di stato della risposta i valori che deve assumere ogni singolo parametro `path`. Prima di spiegare però come questo avviene verrà illustrata l'altra soluzione.

La seconda soluzione consiste, anche in questo caso nell'escludere il generatore, e nel determinare quali siano i valori dei parametri `path` per cui esiste una rappresentazione esplorando il servizio, ovvero cercando di scoprire a priori almeno un valore che genera un success e uno che genera un client error. Riprendendo l'esempio fin qui usato, mettere in pratica questa soluzione per il servizio che gestisce il database di prodotti vorrebbe dire effettuare al massimo dieci chiamate, che di fatto sono quelle riportate nella Tab. 2.4. Nella situazione descritta le chiamate necessarie al fine di scoprire un valore valido e uno non valido sono tre. Infatti, una volta individuato l'intervallo di valori ammessi, si inizierebbe a inviare richieste GET all'endpoint `/products/productId` avendo cura di valorizzare ogni volta `productId` con uno dei valori appartanenti all'intervallo. Quindi, procedendo in maniera ordinata (dal minimo al massimo):

- la prima chiamata con `productId` uguale a 1 tornerebbe 200, pertanto resterebbe da scoprire ancora un valore che genera client error;
- la seconda chiamata con `productId` uguale a 2 tornerebbe 200, pertanto resterebbe ancora da scoprire un valore che genera client error;
- la terza chiamata con `productId` uguale a 3 tornerebbe 404, pertanto si può terminare la fase esplorativa avendo entrambi i valori desiderati.

Ovviamente, questa soluzione diventa leggermente più impraticabile, o quantomeno computazionalmente più pesante, se uno dei due limiti non è finito. Se l'intervallo di validità fissato per `productId` non fosse, per esempio, limitato superiormente, si rischierebbe di effettuare un elevato numero di chiamate prima di trovare i due valori cercati. Si consideri poi che se il tipo di `productId` non fosse `number`, bensì `string`, la soluzione diventerebbe quasi completamente inattuabile.

A questo punto resta da illustrare la prima soluzione, ovvero quella che è stata effettivamente adottata. Come anticipato, consiste nel lasciare che sia l'utente a specificare manualmente i valori da associare ai parametri di tipo `path`. Questa scelta ovviamente riduce in parte il grado complessivo di automatizzazione del tool, in quanto richiede un ulteriore intervento da parte dell'utente. Ad ogni modo, come l'utente interagirà in questa fase sarà spiegato nel manuale utente. Perora basta sapere che all'utente sarà richiesto di compilare un file json che ha una struttura ben precisa, e che, rispecchia la stessa struttura dell'oggetto da passare a swagger-test-templates come parametro `requestData` (Fig. 2.10). In questo caso però i parametri che compaiono dentro ogni oggetto

dell'array associato ad ogni codice di risposta sono parametri di tipo `path` oppure parametri per cui l'utente ha richiesto di specificare manualmente il valore. La scelta che si è fatta, per venire incontro a esigenze particolari (ad esempio dover assegnare un valore preciso a un parametro che dipende dai valori assegnati a degli altri parametri di tipo `path`), è stata quella di permettere all'utente di assegnare dei valori arbitrari a qualsiasi parametro di una terna endpoint-metodo-codice di stato. L'utente dovrà sempre obbligatoriamente fornire i valori da assegnare ai parametri di tipo `path`, e può stabilire per quali altri parametri impedire di fatto che il generatore generi dei valori.

Come questi valori verrano impiegati dall'applicazione verrà spiegato più avanti, perora basti sapere che è possibile inibire l'uso del generatore per alcuni parametri.

Tornando allo script dell'applicazione, se il parametro preso in considerazione non è un parametro di tipo `path` e non è un parametro per cui l'utente ha richiesto l'inibizione della generazione automatica, il generatore viene invocato passando come parametro lo schema JSON, se il parametro è un oggetto, o il tipo primitivo. Il generatore, come più volte detto, restituisce un array di oggetti che contengono delle informazioni sul parametro per cui si è richiesta la generazione e il valore proposto dal generatore stesso. Questi oggetti vengono arricchiti di altre proprietà come ad esempio il nome del parametro e la sua posizione (`body`, `query` ecc.). Inoltre, se il tag `x-example` è presente nella descrizione del parametro che si sta trattando, il valore generato marcato come valido viene sovrascritto con il valore associato al tag. In questo caso si è scelto di introdurre un tag personalizzato seguendo lo schema di naming suggerito nella documentazione di Swagger. In effetti, un tag pensato per questo tipo di situazioni esiste, ed è il tag `example`, tuttavia si può utilizzare solo nella definizione dei tipi e non, per esempio, nella sezione dedicata alla dichiarazione di un parametro. L'uso del tag `example` all'interno della dichiarazione di un parametro causerebbe il lancio di un'eccezione durante il processo di validazione da parte della libreria ZSchema. L'uso del tag personalizzato `x-example` copia quindi ed estende in altri punti della descrizione l'utilizzo del tag `example` che risulta appropriato e segue le linee guida esposte nella documentazione di Swagger. L'alternativa al tag `x-example` potrebbe essere il tag `default`. Tuttavia l'uso del tag `default` per questo scopo è scoraggiato da Swagger poiché dovrebbe essere usato per specificare i valori di default di parametri non obbligatori da inserire nelle richieste HTTP qualora il client decidesse di non fornire il valore per lo specifico parametro non obbligatorio

Al fine di avere un comportamento più generale ed avere una maggiore quantità di dati generati, il generatore è stato modificato rispetto al suo funzionamento di default. Rispetto a quanto spiegato all'inizio del paragrafo riguardo al generatore che sarebbe stato usato, cioè Json-schema-test-data-generator, è stata apportata una modifica. Come detto, di default, il generatore propone un solo valore non corretto per via del tipo errato, come veniva illustrato nella Tab. 2.2. Il tipo errato veniva scelto casualmente dall'insieme di tutti i tipi noti al generatore, privato, ovviamente, del tipo del parametro per cui il generatore stesso era stato invocato. La modifica è consistita nel rimuovere questa limitazione e lasciare che il generatore non si limiti più a proporre un solo valore sbagliato per tipo, e che ritorni n valori errati, ognuno dei quali è di tipo diverso. A titolo di esempio, si consideri ciò che accadrebbe se il generatore, modificato secondo quanto appena esposto, venisse invocato per generare dei valori ricevendo in ingresso un parametro di tipo `String`. Sapendo che, il generatore conosce 5 tipi primitivi, ovvero `String`, `Boolean`, `Number`, `Null` e `Integer`, i valori non validi generati dal generatore sarebbero quattro di tipo `Boolean`, `Number`, `Null` e `Integer`. La Tab. 2.6 illustra i tipi dei valori generati per ogni possibile tipo del parametro di input.

Tipo parametro input	*Tipo errato output*
String	Boolean, Number, Null, Integer
Boolean	String, Number, Null, Integer
Number	String, Boolean, Null, Integer
Integer	String, Boolean, Number, Null

Tabella 2.6. I tipi errati proposti dal generatore per ogni tipo noto.

Dalla Tab. 2.6 si vede come per ogni tipo in ingresso il generatore restituisce sempre `null`. Qui `null` più che un tipo al pari di `String` o `Number` va inteso come valore per indicare assenza. Attraverso il valore `null` il generatore suggerisce all'applicazione chiamante di non assegnare alcun valore al parametro. Ciò si traduce nell'esclusione del parametro dalla specifica chiamata HTTP.

Si consideri nuovamente l'esempio del servizio per la gestione dei prodotti. Supponiamo che l'endpoint `/products` accetti il parametro `numberOfProductsToShow`, un intero opzionale. Attraverso questo parametro il client può chiedere al servizio di mostrare un numero di prodotti pari al valore assegnato al parametro `numberOfProductsToShow`. Essendo opzionale, il client può anche scegliere di omettere il parametro. In questo caso, è verosimile ipotizzare che il servizio restituisca tutti i prodotti presenti nel database, senza applicare alcun filtro sulla quantità di prodotti restituiti.

Con le modifiche applicate al generatore alcuni dei valori proposti potrebbero essere i seguenti:

- ```
  { "valid" : true,
    "data" : 5,
    "message" : "should work with all required properties" }
  ```
- ```
  { "valid": false,
    "data" : "qwerty",
    "message" : "should not work with type 'String' " }
  ```
- ```
  { "valid" : false,
    "data" : 451.26,
    "message": "should not work with type 'Number' " }
  ```
- ```
  { "valid" : false,
    "data": true,
    "message" : "should not work with type 'Boolean' " }
  ```
- ```
  { "valid" : false,
    "data" : null,
    "message" : "should not work with type 'null' " }
  ```

Il generatore non accede direttamente alla descrizione, e non può sapere che il parametro per cui è stato chiamato alla generazione è opzionale. Pertanto marca come valido il primo oggetto creato e come non validi i restanti quattro. Tuttavia marcando l'oggetto con valore `null` come non valido il generatore commette un errore. Il valore `null` come spiegato indica l'assenza del parametro nella chiamata HTTP. In questo caso il parametro `numberOfProductsToShow` è opzionale, pertanto il valore `null` deve essere considerato valido. Come detto il generatore non può sapere che il parametro è opzionale, pertanto sta all'applicazione rimediare e correggere il valore assegnato al campo `valid` dell'oggetto. Chiaramente, il test che viene fatto dall'applicazione per stabilire se attuare la correzione o meno è verificare il valore della proprietà `required` del parametro nella descrizione. Se il parametro è opzionale (`required` è `false`) il campo `valid` dell'oggetto che ha per dato `null` viene invertito, passando da `false` a `true`.

Terminato il ciclo sui parametri dell'endpoint, l'applicazione, che nel frattempo ha raccolto informazioni che saranno utili più avanti come ad esempio il numero di parametri, la loro posizione, e se i valori per ogni singolo parametro sono stati ottenuti dal generatore o meno, dispone, per ogni parametro previsto dall'endpoint, di un insieme di valori, validi e non, generati automaticamente, che devono essere combinati tra di loro in modo da creare quelli che potremmo chiamare "dataset", cioè l'insieme dei valori che devono assumere i parametri per ogni singola richiesta HTTP. In particolare, l'applicazione ha costruito un oggetto (`getMocks`) che ha per chiavi i nomi dei parametri per cui il generatore è stato invocato, e per valori gli array restituiti dal generatore i cui oggetti sono stati arricchiti o modificati come spiegato sopra. In Fig. 2.19 viene riportata la struttura dell'oggetto `getMocks`.

Noti i dati che l'applicazione ha a disposizione e lo scopo finale, la strategia che è stata adottata in questa fase è stata quella di generare prima le combinazioni per gli happy path, e dopo, combinare i dati per le richieste che dovrebbero ritornare un client error, impiegando i valori non validi.

Per generare i dataset relativi agli happy path, si è scelto di combinare i dati in modo da:
  ```

```
{
  "param-1" : [
    {generated-object-1-for-param-1}, {generated-object-2-for-param-1}, ...,
        {generated-object-n-for-param-1}
  ],
  "param-2" : [
    {generated-object-1-for-param-2}, {generated-object-2-for-param-2}, ...,
        {generated-object-n-for-param-2}
  ],
  ...
  "param-m" : [
    {generated-object-1-for-param-1}, {generated-object-2-for-param-2}, ...,
        {generated-object-n-for-param-m}
  ]
}
```

Figura 2.19. La struttura dell'oggetto getMocks.

- ottenere dei dataset che contengano tutti i parametri, obbligatori e non, valorizzati con valori validi;
- ottenere dei dataset che contengano i soli parametri obbligatori valorizzati con valori validi.

È importante ricordare che è in questa fase del flusso di esecuzione che l'applicazione comincia a costruire l'oggetto che verrà poi passato in ingresso a swagger-test-template come valore del parametro `requestData`.

Per quanto riguarda il primo punto, ovvero la generazione dei dataset contenenti sia i parametri obbligatori che quelli opzionali, le operazioni che vengono effettuate sono tecnicamente abbastanza lineari. Per ogni parametro presente in `getMocks`, si filtra il corrispondente array contenente gli oggetti creati dal generatore in modo da tenere soltanto quelli marcati come validi. Da questa operazione di filtro si ottiene un array che può contenere uno o due oggetti, entrambi marcati come validi. L'array conterrà solo un oggetto se il corrispondente parametro è obbligatorio. Il generatore infatti restituisce sempre solo un oggetto valido. Ci saranno invece due oggetti marcati come validi nell'array filtrato se il parametro è opzionale. In questo caso infatti, oltre all'unico oggetto valido restituito dal generatore, bisogna aggiungere anche l'oggetto di tipo `null` che va considerato valido visto il carattere opzionale del parametro. L'operazione da eseguire è quindi quella di scartare, in questa fase, i valori `null` validi, che saranno presi in considerazione successivamente. I parametri opzionali saranno quindi valorizzati con dei valori e non con `null`. A questo punto, ciclando su tutti gli oggetti restituiti dall'applicazione dei filtri, e ricordando la struttura che deve avere l'oggetto da passare al parametro `requestData` di swagger-test-templates riportata anche in Fig. 2.10, si comincia a creare un oggetto che ha:

- la proprietà `body` se il valore generato è relativo al parametro `body`;
- una proprietà per ogni valore valido ottenuto dal filtro, con nome il nome del parametro stesso;
- la proprietà description.

Tutto questo avviene attraverso l'esecuzione del frammento di codice riportato in Fig. 2.20

Il dataset così creato di per sé non ha alcun significato se non associato a una terna endpoint-metodo HTTP-codice di stato della risposta. Il dataset cioè deve essere collocato a un certo punto della gerarchia mostrata in Fig. 2.10. Pertanto è necessario individuare una terna ben precisa. I primi due elementi della terna (endpoint e metodo HTTP), sono ben noti in questo flusso, poiché la

```
var validMocks = {};
Object.keys(getMocks).forEach(function(paramName) {
  getMocks[paramName].filter(function (fullMock) { return fullMock.valid;
      }).forEach(function (mock) {
    if(mock.data != null) {
      if(mock.in == 'body') {
        validMocks['body'] = mock.data;
      }
      else validMocks[paramName] = mock.data;
      validMocks.description = mock.message;
    }
  });
});
```

Figura 2.20. Lo script che genera i dataset con i tutti i parametri.

generazione è scatenata dalla lettura dei parametri dalla descrizione per una certa coppia. Stabilire il terzo elemento della terna è l'operazione più complicata, in generale. Tuttavia in questa fase, in cui si generano gli happy path, individuare il codice di stato della risposta risulta semplice. L'assunzione che si è fatta è dare per scontato che la descrizione dichiari come risposte previste per una certa coppia endpoint-metodo HTTP uno tra i codici di stato di Success più "comuni" (200-OK, 201-Created, 204-No Content).

Giunti in questa fase del flusso di esecuzione, bisogna accertare che il dataset corrente sia completo. È possibile infatti che il dataset sia incompleto, poiché include soltanto i valori da assegnare a quei parametri della richiesta HTTP per cui il generatore è stato invocato. Il dataset, cioè, non include:

- i parametri path (se previsti);
- i parametri per cui l'utente ha esplicitamente inibito l'uso del generatore.

Per completare il dataset quindi è necessario aggiungere i parametri mancanti i cui valori sono reperibili all'interno del file che è stato preparato e opportunamente compilato dall'utente. Indirizzare il file, e individuare i parametri da aggiungere, insieme ai corrispondenti valori, è semplice, poiché la struttura JSON è ben nota. La terna endpoint-metodo HTTP-codice di stato della risposta è stata definita e grazie ad essa è possibile acquisire dal file un array di oggetti che hanno tutti le stesse proprietà. Ogni oggetto, che potremmo definire un dataset da completamento, contiene una proprietà per ogni parametro definito manualmente, con nome il nome del parametro stesso, e per valore quello assegnato dall'utente.

Dato che la terna individua un array, e che potenzialmente l'array può contenere più di un dataset da completamento, l'operazione da svolgere consiste nel duplicare il dataset incompleto corrente, creando tante copie quant'è la dimensione dell'array, e fondere ogni copia ottenuta con ognuno dei dataset da completamento. In questo modo, al termine di questo processo si avrà un numero di dataset completi pari alla dimensione dell'array individuato dalla terna nel file JSON compilato manualmente dall'utente. Tutti i dataset avranno tutti le stesse proprietà. Alcune di queste sono quelle portate in dote dal dataset incompleto, e sono quelle per le quali il generatore è stato invocato. Ovviamente, individuata una di queste proprietà, ogni dataset completo avrà associato a tale proprietà lo stesso valore. La Tab. 2.7 cerca di evidenziare la struttura dei dataset completi che condividono tutti gli stessi valori per le proprietà per cui è stato usato il generatore.

Se invece, la terna corrente non prevede parametri path e l'utente non ha specificato parametri manualmente, il dataset va considerato completo e dunque può essere aggiunto all'oggetto che verrà poi passato in ingresso a swagger-test-template come valore del parametro `requestData`.

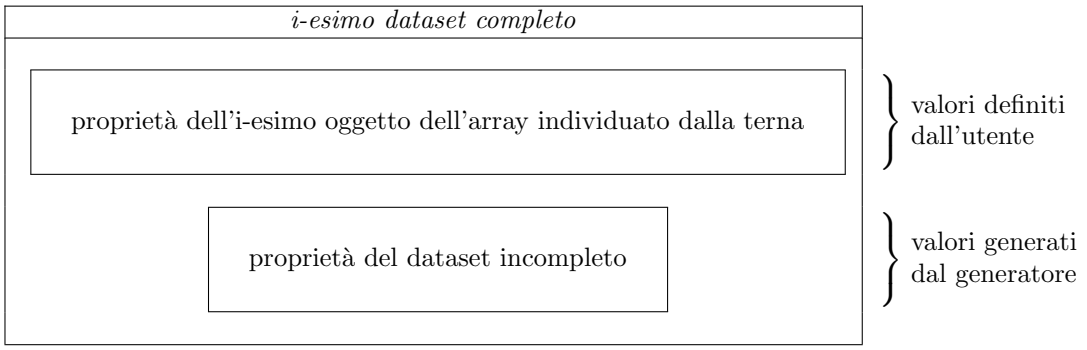

Tabella 2.7. La struttura di un dataset completo.

Al termine di questa fase l'applicazione ha terminato la creazione dei dataset relativamente al primo punto precedentemente descritto riguardo la generazione dei dataset contenenti tutti i parametri, opzionali e non. Quindi, per chiudere la generazione dei dataset per gli happy path, è necessario generare quelli che contengono solo i parametri obbligatori. La scelta che si è fatta è stata quella di considerare soltanto le due situazioni estreme, ovvero, quella in cui tutti i parametri sono valorizzati, e quella in cui i soli parametri obbligatori hanno un valore associato. Ovviamente, l'alternativa poteva essere quella di creare tutte le combinazioni semplici, fissando i parametri obbligatori. Se per esempio, uno specifico endpoint prevede, oltre ai parametri obbligatori, anche tre parametri opzionali, che chiameremo semplicemente A, B e C, le combinazioni sarebbero state otto. La Tab. 2.8 mostra le otto alternative disponibili quando si prova a valorizzare i tre parametri, ricordando che ogni parametro opzionale accetta un valore ma anche il valore null.

A	*B*	*C*
null	null	null
valore	null	null
null	valore	null
null	null	valore
valore	valore	null
null	valore	valore
valore	null	valore
valore	valore	valore

Tabella 2.8. Le alternative disponibili quando si prova a valorizzare i tre parametri.

Tuttavia l'alternativa di creare tutte le possibili combinazioni non è stata messa in atto in quanto si sarebbero ottenuti un elevato numero di dataset validi, e, in generale, perchè si è scelto di porre più attenzione ai dataset non validi, ovvero quelli da associare ai codici di stato della risposta relativi a un client error.

Ad ogni modo, per creare i dataset senza i parametri opzionali, o meglio, con i parametri opzionali valorizzati a null, il procedimento è molto simile a quello esposto in precedenza. In particolare, nella fase precedente, si è tenuto traccia dei parametri obbligatori. In questo modo, sapendo, per ogni parametro, se questo è opzionale o meno, ogni dataset che viene creato avrà:

- le proprietà relative ai parametri obbligatori valorizzate con i valori ottenuti dal generatore;
- le proprietà relative ai parametri opzionali valorizzate a null.

Anche in questo caso, come fatto in precedenza, è necessario verificare che i dataset creati non siano incompleti, ed eventualmente ottenere i dataset completi seguendo la stessa logica adottata in precedenza. In Fig. 2.21 il frammento di codice che crea i dataset desiderati. Si noti l'uso dell'array `requiredParamArray` (righe 3 e 12) che è stato precedentemente creato e, come si può intuire dal nome, contiene i nomi dei soli parametri obbligatori.

```
1  var validMocks = {};
2  Object.keys(getMocks).forEach(function(paramName) {
3    if(requiredParamArray.includes(paramName)) {
4      getMocks[paramName].filter(function (fullMock) { return fullMock.valid;
           }).forEach(function (mock) {
5        if(mock.in == 'body' ) {
6          validMocks['body'] = mock.data;
7        }
8        else validMocks[paramName] = mock.data;
9        validMocks.description = mock.message;
10     });
11   }
12   else {
13     validMocks[paramName] = null;
14   }
15 });
```

Figura 2.21. Lo script che genera i dataset con i soli parametri obbligatori.

A questo punto si chiude quella che potremmo definire la come la prima parte della creazione dei dataset. Finora infatti l'applicazione ha creato solo i dataset associati ai success, e che porteranno alla generazione dei test che verificano il comportamento corretto del servizio. I test generati a partire da questi dataset compiranno delle chiamate HTTP agli endpoint del servizio che sono ben formate ed è dunque ragionevole aspettarsi che il servizio:

- risponda con un success;
- risponda con un corpo che è conforme a quanto precisato nella descrizione.

I punti precedenti corrispondono alle due asserzioni specificate in ogni test generato automaticamente.

Resta quindi ancora da preparare altri dataset che useranno alcuni dei dati marcati come non validi dal generatore, e associarli ai codici di stato della risposta che cadono nell'insieme dei client error. I criteri con cui i valori non validi saranno usati nella creazione dei dataset, e come quest'ultimi saranno associati ai codici di stato della risposta verrà esposto di seguito.

La prima cosa che bisogna sottolineare è che, al fine di mantenere la creazione dei dataset più semplice e strutturata, si è scelto di distinguere due casi:

- il caso in cui l'endpoint accetti solo un parametro;
- e il caso in cui l'endpoint accetti più parametri.

Per questioni di semplicità si esporrà prima il primo caso. Se l'endpoint accetta solo un parametro bisogna distinguere il caso in cui per questo unico parametro è stato invocato il generatore, e il caso in cui il parametro è un parametro di tipo `path` oppure l'utente ha inibito l'uso del generatore.

Se si tratta di un parametro per cui è stato invocato il generatore, le operazioni che si compiono sono molto simili a quelle che venivano svolte nei frammenti di codice precedenti (Fig. 2.20). L'unica differenza è che mentre lì si filtrava l'array `getMocks` alla ricerca dei soli oggetti validi, qui invece l'array viene filtrato escludendo gli oggetti validi. Ogni oggetto presente nell'array così filtrato contribuirà alla creazione di un dataset. Creare il dataset è immediato. Ogni dataset infatti avrà un'unica proprietà, con nome il nome del parametro, l'unico previsto dall'endpoint, che sarà valorizzata con il valore di ognuno degli oggetti presenti nell'array filtrato. Quindi il numero dei

dataset creati in questa fase è pari alla dimensione dell'array filtrato, ovvero al numero di oggetti creati dal generatore e marcati come non validi.

Giunti a questo punto, si ripresenta lo stesso problema riscontrato in precedenza, ovvero come associare i dataset appena creati ai codici di stato della risposta. Nel caso precedente, risolvere questo problema era semplice, perché si stavano creando i test per gli happy path. In questo caso, però, i test che verranno generati a partire da questi dataset, includono chiamate HTTP che dovrebbero causare dei client error. Ad ogni modo, per generare i test è necessario individuare una terna. Chiaramente, individuata la coppia endpoint-metodo HTTP, che anche qui è nota come nei casi precedenti, i codici di stato candidati ad essere associati ad ogni dataset sono quelli dichiarati nella descrizione, in particolare nella sezione `responses`. Quindi, apparentemente, sembrerebbe che l'applicazione debba scegliere uno dei codici ed associare il singolo dataset al codice scelto. Tuttavia come può l'applicazione capire autonomamente quale codice scegliere?

Si consideri per esempio il caso del servizio per la gestione dei prodotti. Nell'esempio precedente si era presentato il caso dell'endpoint `products` aggiungendo l'uso del parametro `numberOfProducts ToShow`. Si supponga adesso di modificare l'endpoint aggiungendo un nuovo parametro di tipo `query` che chiameremo `pin`. Il parametro è una stringa e il servizio risponderà correttamente solo se il client valorizzerà il parametro `pin` con la password corretta, per esempio una specifica sequenza di cinque numeri. Viceversa, in caso di pin errato, il servizio risponderà con il codice di stato 401-Unauthorized. Inoltre il servizio prevede ancora il parametro opzionale `numberOfProducts ToShow`. Nel caso in cui il client valorizzi questo parametro con un valore che non è un intero, come atteso, per esempio con una stringa, il servizio risponderà con il codice di stato 400-Bad request. Quanto detto può essere riassunto nel frammento della documentazione del servizio riportato in Fig. 2.22.

```
/products:
    get:
      description: returns the list of all products
      produces:
        - application/json
      parameters:
        - name: pin
          in: query
          type: string
          required: true
        - name: numberOfProductsToShow
          in: query
          type: integer
          required: false
      responses:
        '200':
          description: returns the list of all products
          schema:
            type: array
            items:
              $ref: '#/definitions/Products'
        '400':
          description: Bad request (wrong format of numberOfProductsToShow)
        '401':
          description: Unauthorized (wrong pin)
```

Figura 2.22. La sezione della descrizione del servizio che gestisce il database di prodotti relativo all'endpoint `products`.

Ancora non è stato spiegato come vengono creati i dataset nel caso in cui la coppia endpoint-metodo HTTP preveda più di un parametro, tuttavia per adesso basta sapere che la logica che è stata applicata porta alla creazione di dataset che hanno solo un parametro non valido. Pertanto, durante il flusso di esecuzione dell'app è verosimile supporre che alcuni dei dataset creati a questo punto siano simili a quelli rappresentati in Fig. 2.23.

```
//dataset 1
{
  "pin" : 9423.45,
  "numberOfProductsToShow" : 5,
  "desciption" : "should not work with pin of type Number"
}
//dataset 2
{
  "pin" : "12345",
  "numberOfProductsToShow" : "lorem ipsum",
  "desciption" : "should not work with numberOfProductsToShow of type String"
}
```

Figura 2.23. Due dataset non validi generati dall'applicazione per l'operazione GET `products`.

Idealmente l'applicazione dovrebbe:

- associare il primo dataset al codice di stato 401, poiché al parametro `pin` è stato assegnato un numero decimale e non un intero;
- associare il secondo dataset al codice di stato 400, poiché al parametro `numberOfProductsToShow` è stato assegnata una stringa e non un intero.

Tuttavia l'applicazione non ha nessun mezzo per compiere autonomamente tali associazioni. L'unico tag all'interno della descrizione che fornisce informazioni utili ad effettuare questo tipo di associazioni è il tag `description` contenuto sotto i codici della risposta nella sezione `responses`. Tuttavia tali informazioni sono espresse in linguaggio umano in quanto il tag `description` è pensato per aggiungere verbosità alla descrizione.

In queste condizione l'unica alternativa che ha l'applicazione è quella di associare ogni singolo dataset alla parola chiave `default`. Quindi il terzo elemento della terna a cui associare i dataset sarà proprio `default`. Spesso nella descrizioni Swagger dei servizi, sotto la sezione delle risposte, è possibile trovare il tag `default` insieme ai tag dei codici della risposta. Questo tag, secondo la documentazione, risulta comodo quando un servizio può rispondere con dei codici di stato diversi ma con la stessa struttura del corpo della risposta. In questo modo si descrivono i possibili errori del servizio collettivamente e non individualmente. In questo caso però il carattere del tag `default` è stato ignorato, e il tag è stato sfruttato semplicemente come parola chiave nella comunicazione con swagger-test-templates e quindi nella generazione dei test.

Inoltre, il comportamento di swagger-test-templates al momento della generazione dei test in presenza della parola chiave `default` è stato modificato rispetto all'originale, per venire in contro alle esigenze dell'applicazione. Normalmente, infatti, swagger-test-templates crea i test per il tag `default` solo se questo compare nella descrizione del servizio. Questo significa che, anche se l'oggetto assegnato a `requestData` in fase di creazione dei test, contiene dei dataset associati ad una terna dove il codice di stato della risposta è `default`, questi vengono ignorati.

Inoltre tutti i template usati da swagger-test-templates, sia quelli relativi alla libreria `request` che quelli che impiegano la libreria `supertest`, contengono una discriminante proprio sul tag `default`. In particolare, il controllo che veniva fatto a tempo di generazione del singolo test, riguardava la sezione relativa alle asserzioni effettuate quando si riceve la risposta del servizio.

Una delle due asserzioni riguarda infatti il codice di stato della risposta. Tuttavia in questo caso swagger-test-templates ha generato un test in cui il terzo membro della terna è `default` e non un vero codice di stato. Per questo, in queste condizioni, quello che succede è che l'asserzione diventa:

```
expect(res.statusCode).to.equal('DEFAULT RESPONSE CODE HERE');
```

In questo caso, swagger-test-templates si limita a creare l'asserzione con il placeholder "DEFAULT RESPONSE CODE HERE" ed invita l'utente a sostituire il placeholder con il vero codice della risposta. Chiaramente, se l'asserzione non venisse modificata dall'utente il test fallirebbe sempre.

Le modifiche che sono state apportate a swagger-test-templates hanno permesso di:

- considerare sempre i dataset contenuti in `requestData` associati alle terne dove il terzo elemento è `default`;
- avere nei test generati per queste terne un'asserzione che non necessita di essere modificata dall'utente prima dell'esecuzione del test.

Questi risultati sono stati ottenuti modificando lo script principale di swagger-test-templates e i file dei template impiegati per la creazione dei test. In particolare, durante il flusso di esecuzione di swagger-test-templates, si leggono i codici dichiarati nella sezione `responses`. Ognuno di questi, insieme all'endpoint e al metodo, individua una terna che viene usata per accedere all'oggetto `requestData` se presente. Pertanto a comandare la generazione dei test sono i codici di stato presenti nella sezione `responses` della descrizione. Questo è il motivo per cui se `requestData` contiene dei dataset associati a terne non note a swagger-test-templates i corrispondenti test non sono generati. Per aggirare questo problema quello che si è fatto è stato forzare la presenza del tag `default`. In questo modo lo script di swagger-test-templates considererà sempre i dataset associati alle terne che hanno come terzo elemento `default`. In pratica si fa credere a swagger-test-template che tutte le coppie endpoint-metodo HTTP prevedano la risposta `default`. Ovviamente, trattandosi di una forzatura, si perde la seconda asserzione che viene solitamente fatta nei test, ovvero quella relativa al corpo della risposta, ma questo accadeva già quando la singola risposta non dichiarava uno schema per la risposta.

L'altra modifica ha interessato i template usati per la generazione dei test. In particolare, l'asserzione riportata in precedenza è stata modificata in modo da verificare che il codice di stato della risposta sia uno tra quelli dichiarato per la coppia endpoint-metodo HTTP nella descrizione. L'asserzione diventa quindi la seguente:

```
expect([200,400,401]).to.include(res.statusCode)
```

In questo caso il test sta verificando che il codice di stato della risposta sia uno tra 200, 400 e 401. Questi codici della risposta non sono scelti casualmente, ma sono quelli dichiarati nella descrizione per la coppia endpoint-metodo HTTP del test. Ovviamente, il template non accede direttamente a queste informazione, che sono invece preparate dallo script principale di swagger-test-templates poco prima di compilare i template stessi.

Quello che in pratica si è fatto è stato generalizzare l'asserzione rendendola più flessibile. Il test infatti non accerterà l'esattezza del codice di stato della risposta, ma si limiterà a verificare che il servizio risponda almeno con un codice che è stato dichiarato nella descrizione. In questo modo, l'asserzione diventa più generale e meno stringente.

Tuttavia, si è anche previsto un meccanismo che permette di mantenere la precisione iniziale dell'asserzione. Tale meccanismo consiste nell'impiego di un tag personalizzato nella descrizione. In particolare, il tag, chiamato `x-onError`, va inserito sotto un qualunque codice di stato nella sezione `responses` di una coppia endpoint-metodo HTTP. L'idea principale è quella di fornire delle informazioni aggiuntive allo script dell'applicazione che consentano una precisa associazione tra i dataset creati, contenenti un solo valore errato, e i codici di stato della risposta.

Riprendendo l'esempio relativo al servizio per la gestione dei prodotti la descrizione del servizio, con l'introduzione del tag `x-onError`, andrebbe modificata. La Fig. 2.24 riporta lo stesso frammento della descrizione del servizio, precedentemente riportata, modificata utilizzando il nuovo tag.

```
/products:
    get:
      description: returns the list of all products
      produces:
        - application/json
      parameters:
        - name: pin
          in: query
          type: string
          required: true
        - name: numberOfProductsToShow
          in: query
          type: integer
          required: false
      responses:
        '200':
          description: returns the list of all products
          schema:
            type: array
            items:
              $ref: '#/definitions/Products'
        '400':
          description: Bad request (wrong format of numberOfProductsToShow)
          x-onError: numberOfProductsToShow
        '401':
          description: Unauthorized (wrong pin)
          x-onError: pin
```

Figura 2.24. La sezione della descrizione del servizio che gestisce il database di prodotti relativo all'endpoint `products`. Si noti l'utilizzo del nuovo tag `x-onError`.

La descrizione così modificata, in particolare la sezione `responses` va letta nel seguente modo:

- una GET all'endpoint `/products` restituisce il codice di stato 400 se il parametro `numberOfProductsToShow` non è valorizzato correttamente;
- una GET all'endpoint `/products` restituisce il codice di stato 401 se il parametro `pin` non è valorizzato correttamente.

L'idea quindi è quella di associare al tag il nome del parametro la cui valorizzazione errata genera in risposta il codice di stato sotto al quale il tag compare. Leggendo la descrizione in questo modo, è la descrizione stessa a fornire all'applicazione delle informazioni utili per effettuare le associazioni tra i dataset e i codici di stato. In pratica, tramite il tag `x-onError` si cerca di introdurre un mezzo per tradurre ciò che veniva espresso in linguaggio umano con il tag `description` e renderlo così accessibile all'applicazione stessa. Inoltre l'applicazione è stata predisposta in modo da poter associare al tag `x-onError` più di un parametro. Quindi al tag è possibile associare sia un solo parametro ma anche un array di parametri. In questo modo si coprono quei casi in cui uno stesso codice di stato può essere provocato da più di un parametro non valorizzato correttamente (non contemporaneamente). Se, per esempio, al tag `x-onError` di un codice di stato `x` è associato un array contenente i parametri `a` e `b` allora l'applicazione ricaverà che il codice di stato `x` viene restituito dal servizio quando il parametro `a` non è valorizzato correttamente oppure quando il parametro `b` non assume un valore accettabile dal servizio.

Tale logica, ricordando che ci troviamo nel caso di endpoint che accetta un solo parametro, è applicata solo nel caso in cui il parametro è stato generato dal generatore. Se questo non è avvenuto,

cioè se il parametro dell'endpoint è un parametro di tipo `path`, l'applicazione va a leggere il valore da assegnare direttamente dal file compilato manualmente. In particolare, tra tutte le terne dove endpoint e metodo HTTP sono fissati, l'applicazione considera solo quella dove il terzo elemento, cioè il codice di stato della risposta è 404, se presente. Solo in questo caso, infatti, è significativo generare un test. Le altre terne devono servire all'applicazione solo da supporto. Devono essere considerate solo per completare eventuali dataset incompleti e non per generarne dei nuovi.

L'ultimo caso che rimane da illustrare è quello in cui si generano i dataset errati per una coppia che accetta più di un parametro. Come anticipato in precedenza, in questa fase l'idea principale è quella di compiere delle operazioni che portano alla creazione di dataset in cui solo un parametro non è valorizzato correttamente. Con questo obiettivo, sapendo che il generatore genera più valori non validi che validi, si può già intuire come il numero di dataset che vengono preparati in questa fase è superiore al numero di dataset validi creati nei flussi precedenti.

Al fine di creare ogni possibile combinazione a partire dai valori creati dal generatore, l'algoritmo, fissato uno degli `n` array di valori generati ne estrapola i soli valori non validi. Per gli altri `n -1` invece ricava i valori marcati come validi. Lo script nella Fig. 2.25 effettua esattamente questo tipo di operazione.

```
Object.keys(getMocks).forEach(function(paramName) {
  var arrayDaPermutare = [];
  var invalidValues = getMocks[paramName].filter(function (fullMock) {
   return !fullMock.valid;})
  arrayDaPermutare.push(invalidValues);
  Object.keys(getMocks).forEach(function(paramName2) {
    if (paramName != paramName2) {
      arrayDaPermutare.push(getMocks[paramName2].filter(function(fullMock) {
          return fullMock.valid;}))
    }
  });
...
```

Figura 2.25. Lo script che prepara i dati che verranno usati nella creazione dei dataset.

Ogni array ricavato seguendo questo procedimento viene salvato all'interno di un altro array che fa da contenitore. Pertanto, al termine del secondo ciclo `for`, l'array contenitore conterrà:

- un array con i valori non validi dell'`n-esimo` parametro;
- `n-1` array ognuno contenente i valori validi dell'`n-1-esimo` parametro.

Questi dati devono essere combinati in modo da ottenere i dataset. Con la logica seguita nel riempire l'array contenitore, la regola per cui ogni dataset conterrà solo un parametro valorizzato con un valore non valido viene rispettata. Ogni array dell'array contenitore infatti contiene i valori di un solo parametro, quindi estraendo un elemento da ogni array dell'array contenitore si ottiene un dataset. L'obiettivo è creare tutte le possibili combinazioni. Per farlo viene invocata la funzione in Fig. 2.26 [5].

Per chiarire che cosa si intende per combinazioni e quindi il comportamento della funzione, viene fatto un esempio. In particolare si mostrerà l'output della funzione quando riceve in ingresso un determinato input campione. Dato che il comportamento della funzione non dipende dal tipo degli oggetti contenuti in ogni array, l'esempio verrà fatto con un input semplificato, ovvero un

[5] https://stackoverflow.com/questions/15298912/javascript-generating-combinations-from-n-arrays-with-m-elements/15300375

array contenitore che contiene array di oggetti generici che verranno rappresentati con le lettere dell'alfabeto. Fig. 2.27 mostra l'output che viene prodotto dalla funzione quando riceve come parametro l'input presente nella stessa figura.

```
function cartesian(arraysToCombine) {
  var divisors = [];
  for (var i = arraysToCombine.length - 1; i >= 0; i--) {
     divisors[i] = divisors[i + 1] ? divisors[i + 1] *
     arraysToCombine[i + 1].length : 1;
  }
  function getPermutation(n, arraysToCombine) {
     var result = [],
         curArray;
     for (var i = 0; i < arraysToCombine.length; i++) {
        curArray = arraysToCombine[i];
        result.push(curArray[Math.floor(n / divisors[i]) % curArray.length]);
     }
     return result;
  }
  var numPerms = arraysToCombine[0].length;
  for(var i = 1; i < arraysToCombine.length; i++) {
      numPerms *= arraysToCombine[i].length;
  }
  var combinations = [];
  for(var i = 0; i < numPerms; i++) {
      combinations.push(getPermutation(i, arraysToCombine));
  }
  return combinations;
}
```

Figura 2.26. La funzione che genera tutte le possibili combinazioni a partire dall'array contenitore.

Come si vede, sia dallo script in Fig. 2.26 che dall'esempio in Fig. 2.27, la funzione restituisce a sua volta un array contenitore. Questo array contiene al suo interno tutte le combinazioni create a partire dall'input passato alla funzione stessa.

```
Input:
[ [A, B, C] , [D, E], [F, G] ]

Output:
[ [A, D, F], [B, D, F], [C, D, F], [A, E, F], [B, E, F], [C, E, F], [A, D,
    G], [B, D, G], [C, D, G], [A, E, G], [B, E, G], [C, E, G] ]
```

Figura 2.27. L'input e l'output della funzione *cartesian()*.

Nello script gli oggetti controllati dalla funzione, in input e in output, sono gli oggetti prodotti dal generatore. Quindi la funzione `cartesian()` ha praticamente già creato i dataset. O meglio in realtà la funzione ha stabilito il numero dei dataset da creare in questa fase e il valore che ogni singolo parametro assumerà in ogni singolo dataset. Quindi, per creare effettivamente il dataset, non resta che ciclare sugli elementi contenuti in ogni singolo array dell'array contenitore restituito dalla funzione `cartesian`, e riempire il dataset seguendo le stesse logiche già applicate nella fase in cui si sono creati i dataset per gli happy path.

Anche in questo caso, una volta creato il singolo dataset, per effettuare l'associazione tra il dataset e la terna endpoint-metodo-codice della risposta si utilizza lo stesso approccio esposto in precedenza nel caso dei dataset con un solo parametro. Di conseguenza anche qui si va alla ricerca del tag `x-onError` per stabilire se associare il dataset a uno specifico codice di stato della risposta o al generico `default`.

Infine, una volta effettuata l'associazione, va verificato se il singolo dataset creato è completo o meno. Nel caso in cui non lo fosse viene completato aggiungendo i parametri e i rispettivi valori leggendoli direttamente dal file che è stato compilato manualmente dall'utente.

Queste ultime operazioni sono identiche a quelle che vengono svolte nei casi di happy path e dataset errato con un solo parametro esposte in precedenza.

A questo punto, il flusso di esecuzione della funzione che genera i dataset per una determinata coppia endpoint-metodo HTTP può terminare. Prima di ritornare però, tutti i dataset generati vengono salvati in un file chiamato `request-array.json` nella directory *./metodoHTTP/endpoint*. Salvato il file, in maniera sincrona, la funzione ritorna l'oggetto `pathmock` che è stato via via costruito e che ha la struttura riportata in Fig. 2.10. Una volta che la funzione ha terminato e ritornato `pathmock` si hanno tutte le informazioni necessarie per lanciare swagger-test-templates. In Fig. 2.28 le righe che preparano l'input e invocano la funzione `testGen` di swagger-test-templates.

```
 var testConfig = {
 assertionFormat: 'expect', testModule: 'request', pathName: [], requestData:
     pathmock, maxLen: -1, templatesPath: templatesPath};
stt.testGen(spec, testConfig).forEach(function(file) {
  fs.writeFileSync(path.join(outputBase, file.name.replace('test', 'spec')),
      file.test);
});
```

Figura 2.28. Le istruzioni che invocano swagger-test-templates e portano alla creazione dei test.

Il modulo `testGen` viene invocato in modo da (l'ordine dei punti seguenti corrisponde ad ognuna delle proprietà di `testConfig`):

- usare le asserzioni nello stile `expect`;
- usare la libreria `request` per fare le richieste HTTP;
- creare i test per tutti gli enpoint;
- creare i test con i dataset contenuti in `pathmock`;
- non imporre alcuna limitazioni in termini di lunghezza della descrizione letta;
- usare i template di default.

La funzione `testGen` come già spiegato nel paragrafo dedicato a swagger-test-templates restituisce delle coppie che hanno per elementi il nome del test generato e il codice sorgente del test stesso. Quindi il test di ogni coppia viene salvato nella directory specificata dall'utente avendo cura di sostituire la stringa "test" con la stringa "spec" nel nome che sarà dato al test stesso. Tale accorgimento è importante poiché per eseguire i test con il comando `mocha` il nome del file deve terminare col suffisso "-spec.js". Per quanto riguarda invece la prima parte del nome di ogni test, viene scelto il nome dell'endpoint, del path relativo. Tuttavia non potendo usare come caratteri lo slash (/) lo si sostituisce con il trattino (-). Queste ultime operazioni vengono effettuate internamente da swagger-test-templates.

Questa è una delle ultime operazioni che compie swagger-test-templates prima che la funzione `testGen` ritorni. I passaggi che vengono effettuati internamente dalla funzione `testGen` non sono

stati esposti nella Sez. 2.3 in cui si è invece posta l'attenzione al carattere generale del tool dal punto di vista dell'utente, mettendo in risalto le caratteristiche principali delle interfacce esposte dal modulo e al loro utilizzo. Pertanto non si è ancora spiegato il funzionamento interno generale del modulo stesso. Trattandosi di un modulo preesistente sviluppato da terze parti, e non sviluppato nell'ambito dell'implementazione del progetto finale, si chiariranno soltanto alcuni passaggi importanti al fine di comprendere in linea generale quali sono gli step che portano alla creazione dei test a partire dalla descrizione del servizio.

Di fatto l'architettura su cui si basa swagger-test-templates è costituita da una serie di funzioni che vengono invocate in modo da creare una gerarchia di chiamate. Ognuna di queste funzioni è specializzata nella lettura di una sezione specifica della descrizione.

Il punto di ingresso è come più volte detto la funzione `testGen`. Questa si occupa principalmente di verificare gli input con cui è stata invocata e invocare a sua volta la funzione `testGenPath` su tutti i path o solo su quelli specificati dall'utente nella variabile `pathName`. Il ruolo di `testGenPath` consiste nel richiamare a sua volta la funzione `testGenOperation` per ogni metodo dell'endpoint per cui è stata invocata. Prima però effettua alcune operazioni preliminari che porteranno alla preparazione di alcuni dati che costituiranno una parte dei test che è condivisa tra tutti.

La creazione dei test è anch'essa basata su una visione modulare. In questa fase infatti viene compilato un primo template, diverso da quelli presentati in precedenza. Questo template, chiamato `outerDescribe.handlebars` contiene la struttura più esterna che condivideranno i test. In particolare, include le import necessarie affinché il test funzioni correttamente. Alcune di queste import sono sempre presenti, come ad esempio l'import della libreria `ZSchema`, altre sono condizionate dalla configurazione ricevuta in ingresso. Ad esempio delle librerie `request` e `supertest` solo una sarà importata, e sarà quella che l'utente ha specificato nell'oggetto di configurazione passato a `testGen`. La stessa logica si applica per le librerie `expect`, `should` e `assert`.

Già nelle funzione precedente, `testGen`, altri due template erano stati preparati. Si tratta di:

- schema.handlebars;
- environment.handlebars.

Questi due template sono molto semplici e si occupano di gestire due aspetti importanti.

Il template `schema`, come si può dedurre dal nome, gestisce le dichiarazioni degli schemi JSON. L'intero template è costituito da una singola riga:

```
var schema = {{schema}};
```

Questo template viene compilato quando è necessario dichiarare nei test schemi JSON. Questo avviene, ad esempio, all'interno di ogni singola unità di test per dichiarare lo schema del corpo della risposta prevista per la specifica terna endpoint-metodo-codice di stato. Questo schema è quello che viene usato nella completion della richiesta HTTP per effettuare l'asserzione sul corpo della risposta.

L'altro template, `environment.handlebars`, è invece usato per gestire delle variabili d'ambiente che solitamente sono condivise tra tutti i test che vengono generati. In Fig. 2.29 è riportato il template `environment.handlebars`.

```
{{#if envVars}}
{{#each envVars}}
{{this}}=YOUR_TOKEN_GOES_HERE
{{/each}}{{/if}}
```

Figura 2.29. Il template environment.handlebars.

Lo scopo principale per cui questo template è impiegato è la gestione dei livelli di sicurezza e autenticazione nelle chiamate di ogni unit test, ovvero gestire il contenuto che nelle descrizioni Swagger appare in corrispondenza del tag `securityDefinitions`. La versione 2.0 di Swagger riconosce fino a tre definizioni di sicurezza, ovvero:

- HTTP Basic Auth;
- API Key;
- OAuth 2.

Per mantenere la gestione dei token centralizzata e fare in modo che tutti i test possano accedere alle stesse informazioni, se la descrizione prevede un qualunque schema di autenticazione, viene generato un file di environment `.env` che contiene una lista di coppie in cui il primo elemento identifica il token, il secondo è il valore associato al token stesso. All'interno poi dei template relativi ai singoli metodi HTTP vi sono dei riferimenti ai singoli token del file `.env`. I token vengono letti in base allo schema di autenticazione previsto per il singolo endpoint o per tutti gli endpoint del servizio. Se il servizio non prevede alcuno schema di autenticazione il file `.env` non viene generato e contestualmente nel singolo test non vi è alcun riferimento ai token di sicurezza.

L'ultimo template utilizzato è chiamato `innerDescribe.handlebars` ed è compilato dalla funzione `testGenOperation`. Anche questo template è costituito da poche righe come si vede dalla Fig. 2.30.

```
describe('{{description}}', function() {
{{#each tests}}
  {{this}}
{{/each}}
});
```

Figura 2.30. Il template innerDescribe.handlebars.

Per capire come il template `innerDescribe` si collega al template `outerDescribe` bisogna aggiungere che la parte finale di quest'ultimo è identica al template appena mostrato in Fig. 2.30.

La Fig. 2.31 mostra lo schema con cui i vari template sono incapsulati. Al livello più esterno c'è il template `outerDescribe`. Ogni template `outerDescribe` che viene compilato corrisponde a un file di test che viene generato. Ogni test è relativo ad uno specifico endpoint esposto dal servizio, infatti il template `outerDescribe` contiene un unica funzione `describe` che ha come argomento il nome dell'endpoint che testa. Il template `outerDescribe` è poi compilato con più template di tipo `innerDescribe`. Ogni template `innerDescribe` ha il compito di descrivere un metodo relativo all'endpoint e contiene quindi un'unica funzione `describe` che prende come argomento il metodo dell'endpoint che descrive. Infine, ogni template `innerDescribe` include a sua volta più unità di test. Queste unità di test coincidono con il risultato della compilazione dei template relativi ai singoli metodi HTTP e sono ottenute a partire dai singoli dataset contenuti in `requestData`. Questi ultimi template sono tutti costituiti da un'unica funzione `it` che segna l'inizio dell'unità.

Riprendendo il flusso che porta alla generazione dei test la funzione a cui viene passato il controllo è la `testGenContentTypes` che si occupa di stabilire per ogni metodo dell'endpoint:

- il tipo della rappresentazione prodotta;
- il tipo della rappresentazione consumata;
- lo schema di sicurezza adottato.

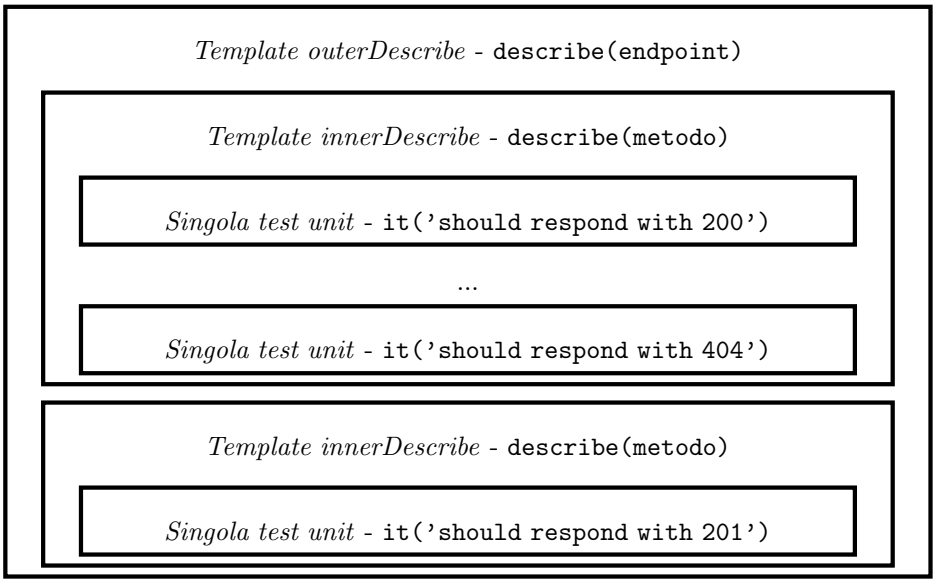

Figura 2.31. Lo schema con cui vengono incapsulati i template.

Successivamente si invoca la funzione `testGenResponse` che è l'ultima funzione della gerarchia. In realtà `testGenResponse` si serve internamente di una funzione, chiamata `getData` che ha il compito di estrapolare dall'oggetto `requestData` i dataset, qualora presenti. Poi `testGenResponse` si occupa, una volta estratti i dataset, di preparare i dati che saranno usati per compilare i template che nello schema di incapsulamento occupano lo strato più interno, ovvero i template relativi ai singoli metodi HTTP.

Per dare un'idea di come si presentano i template relativi ai singoli metodi HTTP si riporta in Fig. 2.32 un frammento del template relativo al metodo GET. Il frammento riporta soltanto la sezione relativa alla creazione della richiesta HTTP. Si noti ad esempio alla riga 12 la scelta del metodo GET. Alle riga 7, invece, viene riempito l'oggetto che conterrà i parametri passati come queryString. Tra questi vi sono anche le chiavi API Key, nel caso il servizio preveda quel tipo di autenticazione. La stessa logica viene applicata per gli header, riga 13. Anche qui sono previste delle istruzioni che tengono in considerazione gli schemi di sicurezza. Oltre allo schema API Key si prevede anche la Basic Auth (header `Authorization` alla riga 18). Come si vede dalle istruzioni a queste righe, per riempire i parametri di sicurezza con i token corretti si accede al file `.env` tramite l'istruzione `process.env.` seguita dall'identificativo nel file del token.

I vari template sono diversi tra loro, in quanto ognuno è specializzato in uno specifico metodo HTTP. Tuttavia alcune parti compaiono in gran parte di essi, come ad esempio il frammento riportato in Fig. 2.32. Altri template poi includono delle sezioni che non appaiono negli altri template. Ad esempio i template per POST e PUT includono una sezione dedicata a prevedere il corpo della richiesta. Il template per il metodo HEAD invece, per esempio, ha solo un'asserzione nella completion della richiesta HTTP che verifica il codice di stato della risposta. Trattandosi del metodo HEAD infatti la corrispondente richiesta HTTP non riceve un corpo e dunque manca l'asserzione sul corpo della risposta.

Infine il modulo è completato da una serie di funzioni di supporto che vengono impiegate direttamente nei template e sono organizzate nel file `helpers.js`.

2.4.4 Il manuale dell'utente

Per eseguire l'applicazione è necessaria la piattaforma Node.js. I vari moduli usati infatti sono tutti scritti in Javascript e si installano col comando `npm`. Node.js è un ambiente costruito sul motore

```
1  request({
2    url: '{{pathify path pathParams}}',
3    {{#isJsonRepresentation contentType returnType}}
4    json: true,
5    {{/isJsonRepresentation}}
6    {{#ifCond queryParameters queryApiKey}}
7    qs: {
8      {{#if queryApiKey}}{{queryApiKey.type}}:
           process.env.{{queryApiKey.name}}{{#if queryParameters}},
9      {{/if}}{{/if}}{{#if queryParameters}}{{#each
           queryParameters}}{{this.name}}: {{requestDataParamFormatter this.name
           this.type ../requestParameters}}{{#unless
           @last}},{{/unless}}{{/each}}{{/if}}
10   },
11   {{/ifCond}}
12   method: 'GET',
13   headers: {
14     'Content-Type': '{{contentType}}'{{#if headerParameters}},
15     {{#each headerParameters}}'{{this.name}}': {{requestDataParamFormatter
           this.name 'string' ../requestParameters}}{{#unless @last}},
16     {{/unless}}{{/each}}{{/if}}{{#if headerApiKey}},
17     {{headerApiKey.type}}: process.env.{{headerApiKey.name}}{{/if}}{{#if
           headerSecurity}},
18     Authorization: '{{headerSecurity.type}} ' +
           process.env.{{headerSecurity.name}}{{/if}}
19     }
20   }
```

Figura 2.32. Il frammento del template GET relativo alla creazione della richiesta HTTP.

JavaScript V8 di Chrome, utilizza un modello I/O non bloccante e ad eventi e il suo ecosistema dei pacchetti, npm, è considerato il più grande ecosistema di librerie open source al mondo[11]. Tramite il sito https://nodejs.org/it/ è possibile scaricare la piattaforma. Al termine dell'installazione, per verificare che Node.js sia stato installato correttamente, è possibile lanciare da terminale il comando `node -v`, che restituisce il numero di versione installata. Se il comando non viene riconosciuto bisognerà aggiungere l'apposita variabile d'ambiente. Successivamente è possibile installare i vari moduli. Per installare `open-api-test-generator` bisogna lanciare il comando:

```
npm install open-api-test-generator
```

Per installare `swagger-test-templates` bisogna lanciare il comando:

```
npm install swagger-test-templates
```

Infine per installare il modulo del generatore si lancia il comando:

```
npm install json-schema-test-data-generator
```

Successivamente è necessario installare i moduli per eseguire i test. I moduli sono `chai`, `ZScema`, `request` e `dotenv`. I corrispondenti comandi che li installano sono:

- `npm install chai`
- `npm install z-schema`

- `npm install request`
- `npm install dotenv`

Infine è necessario sostituire le directory di `open-api-test-generator`, `swagger-test-templates` e `json-schema-test-data-generator` con le rispettive versioni modificate. Di default le directory dei moduli installati con `npm` sono localizzate all'interno della cartella `node_modules` che si trova nella directory dell'utente. Se quando vengono lanciati i comandi per generare o eseguire i test si presenta un errore che informa che un certo modulo non è stato trovato, è necessario installarlo col comando `npm install` seguito dal nome del modulo mancante.

Come si può intuire a questo punto della trattazione, l'intera applicazione funziona tramite linea di comando. Prima di lanciare però lo script dell'applicazione principale è possibile lanciare un secondo script. Questo si occupa di creare un file JSON che contiene l'oggetto che l'utente deve compilare. Più volte in precedenza si è fatto riferimento a dei parametri i cui valori devono o possono essere forniti manualmente dall'utente. Attraverso questo script si rende più semplice e più comodo questo processo. Lo script andrebbe lanciato solo se la descrizione del servizio ha degli endpoint che prevedono dei parametri di tipo `path`. Ciò che fa lo script è creare lo scheletro dell'oggetto JSON che verrà poi eventualmente letto dall'applicazione principale. Per questo motivo il file dello script è stato chiamato `path_param_skeleton_builder.js`. Per creare lo scheletro, per ogni metodo di ogni endpoint della descrizione si verifica se sono presenti parametri di tipo `path`. Se sì, si crea lo scheletro, relativamente alla coppia, rispettando la struttura già mostrata in Fig. 2.10. In Fig. 2.33 è riportata la parte principale dello script.

Come si vede lo script cicla sui metodi di ogni endpoint e per ogni coppia verifica la presenza di parametri `path`. Inoltre lo script verifica un'altra condizione (`desiredParams.includes(param.name)`). L'array `desiredParams` contiene i nomi dei parametri per cui l'utente ha richiesto l'inibizione del generatore. Questo array viene riempito nella prima fase dello script direttamente dall'utente tramite standard input. Quando lo script viene lanciato infatti, questo rimane in attesa dopo aver stampato sulla console il messaggio:

> "Inserisci i nomi dei parametri da aggiungere allo scheletro Json. Termina con una riga vuota"

L'utente può quindi scrivere i nomi dei parametri che desidera avere nello scheletro. Ogni parametro deve occupare una riga. Quando l'utente ha terminato può inserire una riga vuota. Questa operazione scatena la generazione dello scheletro.

Di default lo script associa ad ogni terna un array che contiene un solo oggetto. Nel paragrafo precedente, però si è detto che lo script è stato configurato per aspettarsi anche più di un oggetto all'interno degli array associati alle varie terne. Sta quindi all'utente, se lo desidera e lo ritiene necessario, aggiungere manualmente altri oggetti ai vari array.

Ad esempio, è possibile che ci si ritrovi a testare un servizio con un endpoint che prevede più di un parametro path. Si pensi ad esempio a come apparirebbe l'endpoint di un servizio che implementa una sorta di social network che consente di accedere a uno specifico post di uno specifico utente. L'endpoint sarebbe simile a:

```
/users/{userId}/posts/{postId}
```

Eseguendo lo script che genera lo scheletro la terna relativa al codice di stato 404 sarebbe per esempio:

```
"/users/{userId}/posts/{postId}" : {
  "get" {
    ...
    "404": [ {
      "userId" : "404_userId",
      "postId" : "404_postId"
```

```
...
var pathMocks = {};
for (var localPath in spec.paths) {
  for (var operation in spec.paths[localPath]) {
    var array_parameters = spec.paths[localPath][operation].parameters;
    var pathParams = [];
    if(array_parameters && array_parameters.length > 0) {
      array_parameters.forEach(function(param) {
        if(param.in == 'path' || desiredParams.includes(param.name)) {
          pathParams.push(param.name);
        }
      });
      if(pathParams && pathParams.length > 0 ) {
        if (!pathMocks[localPath]) { pathMocks[localPath] = {};}
        pathMocks[localPath][operation] = {};
        var array_responses = spec.paths[localPath][operation].responses;
        if (array_responses['default'] == undefined) {
          array_responses['default'] = {}
        }
        for(code in array_responses) {
          pathMocks[localPath][operation][code] = [];
          var paramValuesObj = {};
          pathParams.forEach(function (paramName) {
            paramValuesObj[paramName] = code + '_' + paramName;
          });
        pathMocks[localPath][operation][code][0] = paramValuesObj;
        }
      }
    }
  }
}
...
```

Figura 2.33. Un frammento dello script `path_param_skeleton_builder.js`.

```
    } ] ,
    ...
  }
}
```

In questo caso però risulterebbe significativo testare l'endpoint del servizio in due casi, ovvero:

- il caso in cui `userId` è valorizzato con un valore che non identifica alcun utente sul servizio;
- il caso in cui `userId` è valorizzato con un valore che identifica uno specifico utente sul servizio, ma `postId` non identifica alcun post per quell'utente.

In questo modo si otterrebbe sempre il codice di stato della risposta 404. Tuttavia nel primo caso il codice sarebbe provocato dal fatto che non esiste alcun utente con id `userId`. Nel secondo caso, invece, sebbene `userId` identifichi un utente, il codice di stato di "Not found" è causato dal parametro `postId`. Per far sì che l'applicazione generi quindi due test che verifichino questi due comportamenti, nello scheletro l'utente dovrà aggiungere manualmente all'array della terna (/users/userId/posts/postId, GET, 404) un secondo oggetto.

Inoltre dal codice dello script in Fig. 2.33 e dall'esempio appena fatto si vede come ai parametri venga assegnato un placeholder (il codice di stato della terna concatenato al nome del parametro da un underscore). Ovviamente questi placeholder vanno sostituiti con i valori che si vuole che i parametri assumano.

Per eseguire lo script bisogna lanciare da terminale il comando `node` sul file dello script. Inoltre lo script necessita di due parametri aggiuntivi. Si tratta, nell'ordine:

- del path, relativo o assoluto, che individua la descrizione Swagger del servizio;
- della directory in cui si desidera salvare lo scheletro prodotto dallo script.

Entrambi i parametri sono obbligatori. Il comando completo, dunque, necessario per lanciare lo script assume la seguente forma:

```
node path_param_skeleton_builder.js $descriptionPath $destinationPath
```

Entrambi i parametri sono letti impiegando l'interfaccia di Node `process.argv`.

Come già anticipato brevemente in precedenza, l'esecuzione di questo script prima del lancio dell'applicazione non è indispensabile. Lo script che genera lo scheletro JSON risulta però di grande comodità quando si ha a che fare con servizi che prevedono un gran numero di endpoint, metodi e codici di stato della risposta. Inoltre lo script genera un JSON che è ben formato, e lascia all'utente il solo compito di specificare i valori dei vari parametri nelle diverse terne. Tecnicamente, l'utente può decidere di non eseguire lo script e creare il JSON da zero in ogni sua parte, avendo cura di validarlo e di inserire in corrispondenza di ogni terna tutti i parametri `path` previsti.

Anche lo script dell'applicazione principale va eseguito con il comando `node`. Lo script che lancia l'applicazione è stato rinominato `test_gen.js`. In ingresso accetta tre parametri:

- il path che individua la descrizione Swagger del servizio per il quale si vogliono generare i test;
- la directory che dovrà contenere i test generati da swagger-test-templates;
- il path che individua il file con lo scheletro JSON contenente i parametri path.

I primi due parametri sono obbligatori. Il terzo, invece, è opzionale, in quanto è possibile che il servizio non preveda parametri di tipo `path` e che l'utente non abbia specificato alcun parametro per cui escludere il generatore. Anche in questo caso i parametri sono dati in input usando l'array `argv`.

Prima di eseguire i test potrebbe essere necessario riempire correttamente il file di environment create durante la generazione dei test. Se il servizio prevede degli schemi di sicurezza il file `.env` conterrà delle righe come ad esempio:

```
BASIC_AUTH=YOUR_TOKEN_GOES_HERE
```

In questa riga di esempio il file `.env` contiene l'entry relativa al token da usare per la basic auth. Il nome alla sinistra dell'uguale cambia a seconda del tipo di token che identifica e questo dipende dagli schemi di autenticazione riportati nella descrizione del servizio.

Per eseguire i test è necessario il framework `mocha` [12]. Il framework si installa attraverso il comando:

```
npm install mocha
```

Per eseguire i test basta lanciare il comando `mocha` seguito dal percorso del test che si desidera lanciare. Quando si esegue un file di test, le singole unità di test vengono eseguite, e al termine, `mocha` riporta l'esito di ogni singola unità di test. Le singole unit test vengono identificate tramite l'argomento che riceve la funzione `it` che introduce l'unit test. Inoltre per le unit test che falliscono `mocha` riporta il motivo del fallimento, ovvero quale tra le asserzioni presenti non è stata verificata.

Capitolo 3

Risultati

Al fine di testare l'applicazione sono stati realizzati appositamente due servizi relativamente semplici. Entrambi i servizi sono stati scritti in Java usando la libreria Jersey[13], uno dei più importanti framework che implementano l'API JAX-RS (Java API for RESTful Web Services)[14]. Si tratta di una API che permette di implementare servizi web seguendo l'architettura REST. Per farlo JAX-RS fa un vasto uso delle annotazioni Java. L'idea principale è quella di impiegare le annotazioni per vari scopi. Nella Tab. 3.1 vengono riportate alcune delle più comuni annotazioni presenti in JAX-RS.

Nome annotazione	*Descrizione*
@Path	Specifica un endpoint del servizio
@Produces	Specifica il media type della risposta
@Consumes	Specifica il media type della richiesta
@PathParam	Marca un parametro come parametro di tipo path
@HeaderParam	Marca un parametro come parametro dell'header

Tabella 3.1. Alcune annotazioni comunemente usate in JAX-RS.

Per entrambi i servizi si è partiti dalle corrispondenti descrizioni Swagger. A partire da queste poi si sono ottenuti i progetti base in JAX-RS autogenerati tramite la piattaforma SWAGGERhub. Tale piattaforma, come brevemente anticipato in precedenza, offre diverse funzionalità relative alla descrizione di un servizio. In particolare, oltre a fornire un editor per comporre una descrizione e offrire una rappresentazione visuale dell'API descritta, permette anche di generare il codice, in vari linguaggi di programmazione, di server e client. Chiaramente sta all'utente completare i progetti con l'implementazione. SWAGGERhub fornisce quindi solo il punto di partenza. Per esempio, nel caso del progetto di base del server, ogni metodo che descrive il comportamento di un singolo endpoint ha il corpo vuoto e un commento che invita l'utente ad aggiungere l'implementazione.

Il primo servizio è un servizio molto semplice che espone solo due endpoint. Si tratta di un servizio che fa operazioni matematiche banali e restituisce il risultato delle operazioni. L'operazione che svolge il servizio è un'operazione di divisione. Ogni endpoint prevede due metodi: GET e POST. I due endpoint esposti sono dal punto di vista dell'interfaccia identici, tuttavia differiscono nell'implementazione. I due endpoint sono stati chiamati:

- /divisione_ok;
- /divisione_ko.

L'implementazione del primo prevede dei controlli sugli input e quindi prima di procedere con la divisione verifica che l'operazione è fattibile. Per esempio, si verifica che gli input non siano nulli o che il divisore non sia zero. Il secondo endpoint invece non svolge alcun tipo di verifica preliminare

sugli input e prova a svolgere l'operazione immediatamente. Da qui la scelta di associare al primo il suffisso "ok" e al secondo il suffisso "ko".

Per passare al servizio dividendo e divisore sono state pensate due alternative, che coincidono poi con i metodi HTTP esposti da entrambi gli endpoint. Il metodo GET di entrambi gli endpoint prevede due parametri di tipo `query`, chiamati banalmente `a` e `b`. Si tratta di `number` ed entrambi sono parametri obbligatori. Il metodo POST prevede invece che al servizio venga inviato un oggetto JSON con due proprietà anche in questo caso obbligatorie e di tipo `number` e chiamate come nel caso precedente `a` e `b`.

Per quanto riguarda le possibili riposte per le quattro coppie ne sono previste due, ovvero:

- 200 - OK;
- 400 - Bad request.

In quest'ultimo caso è anche previsto che il servizio inserisca nel corpo della risposta un oggetto JSON contenente il codice dell'errore e un messaggio aggiuntivo che descrive l'errore stesso.

Le caratteristiche esposte finora compongono la descrizione del servizio come mostrato parzialmente in Fig. 3.1.

```
"paths": {
  "/divisione_ko": {
    "get": {
      ...
      "parameters": [
        { "name": "a", "in": "query", "description": "il numero a",
          "required": true, "type": "number" },
        { "name": "b", "in": "query", "description": "il numero b",
          "required": true, "type": "number" }
      ],
      "responses": {
        "200": {
          "description": "ok!",
          "schema": { "type": "number" }
        },
        "400": {
          "description": "Bad request", "x-onError" : ["a", "b"],
          "schema": { "$ref": "#/definitions/Errore" }
        }
      }
    },
    "post": {
      "parameters": [
        { "in": "body", "name": "body", "required": true,
          "schema": { "$ref": "#/definitions/Numeri" } }
      ],
      ...
    }
  }
}
```

Figura 3.1. Alcuni parti della descrizione Swagger del servizio per le divisioni.

La descrizione relativa all'endpoint `/divisione_ok`, non riportata in Fig. 3.1, è identica a quella dell'altro endpoint. Inoltre, gli schemi JSON (Numeri e Errore) a cui si fa riferimento sono contenuti nella sezione `definitions` della descrizione stessa.

Lanciando l'applicazione che genera i test, passando come parametro il file contenente la descrizione del servizio delle divisioni, vengono generati complessivamente 32 test, 16 per l'endpoint `divisione_ok` e 16 per l'endpoint `divisione_ko`. Per ogni endpoint, 7 sono relativi al metodo GET e i restanti 9 si riferiscono al metodo POST. La Tab. 3.2 riassume le chiamate effettuate e le corrispondenti risposte ricevute per ogni test che viene generato e il relativo esito relativamente all'endpoint `divisione_ok`.

# Test	*Metodo*	*Parametri*	*Risposta*	*Risposta attesa*	*Esito*
1	GET	?a=-4839474.238726631& b=34757000.365471005	200	200	Superato
2	GET	?a=false& b=34757000.365471005	400	400	Superato
3	GET	?a=null& b=34757000.365471005	400	400	Superato
4	GET	?a=e]gC8o[& b=34757000.365471005	400	400	Superato
5	GET	?a=-4839474.238726631& b=true	400	400	Superato
6	GET	?a=-4839474.238726631& b=null	400	400	Superato
7	GET	?a=-4839474.238726631& b='LugBnJaPE'	400	400	Superato
8	POST	{"a":54416404.07834929, "b":92665777.25740853}	200	200	Superato
9	POST	{"b":-32693045.798083812}	400	200 o 400	Superato
10	POST	{"a":-64274700.92440388}	400	200 o 400	Superato
11	POST	{"a":true, "b":89835092.0006406}	400	200 o 400	Superato
12	POST	{"a":null, "b":-47041982.627544954}	400	200 o 400	Superato
13	POST	{"a":"e)Mq[NUX&0", "b":65483437.360840976}	400	200 o 400	Superato
14	POST	{"a":87039264.24725434, "b":false}	400	200 o 400	Superato
15	POST	{"a":-13116655.045742482, "b":null}	400	200 o 400	Superato
16	POST	{"a":817268.8416136652, "b":"XFZOs"}	400	200 o 400	Superato

Tabella 3.2. I risultati dei test relativi all'endpoint `divisione_ok`.

Come si vede dalla Tab. 3.2 l'applicazione, per ogni metodo, genera sempre un test (test 1 e 8) che effettua una chiamata in cui i parametri rispettano le specifiche della descrizione e quindi ci si attende un success dal servizio. Per quanto riguarda i test 2, 3, 4, 5, 6 e 7 ci si aspetta un client error, in particolare un codice di stato della risposta pari a 400 in quanto i valori assegnati ai parametri non sono dei `number`. L'asserzione in questo caso è esatta in quanto, come mostrato in precedenza nella descrizione, è stato specificato il tag `x-onError` in corrispondenza del codice 400 delle GET per entrambi gli endpoint. L'asserzione relativa al codice di stato della risposta, invece, nei test 10, 11, 12, 13, 14, 15 e 16 è del tipo:

```
expect([200,400]).to.include(res.statusCode)
```

Questi test non contengono l'asserzione sullo schema della risposta. Al contrario gli altri test la contengono. In particolare, i test che hanno come risposta attesa 200 verificano che la risposta contenga un `number`, mentre i test che attendono 400 verificano che la risposta sia conforme allo schema che descrive l'errore. Queste asserzioni sono state sempre verificate.

# Test	Metodo	Parametri	Risposta	Risposta attesa	Esito
1	GET	?a=50321214.15558869& b=79709087.98368526	200	200	Superato
2	GET	?a=true& b=79709087.98368526	400	400	Superato
3	GET	?a=null& b=79709087.98368526	500	400	Fallito
4	GET	?a=Q]uszBT@lAabK3eOf& b=79709087.98368526	400	400	Superato
5	GET	?a=50321214.15558869& b=true	400	400	Superato
6	GET	?a=50321214.15558869& b=null	500	400	Fallito
7	GET	?a=50321214.15558869& b=aFlZBwNeCQY*k[S	400	400	Superato
8	POST	{"a":-88688722.57745072, "b":-39672877.812953725}	200	200	Superato
9	POST	{"b":-69256430.38603231}	500	200 o 400	Fallito
10	POST	{"a":37311488.56068018}	500	200 o 400	Fallito
11	POST	{"a":false, "b":94110355.4795784}	500	200 o 400	Fallito
12	POST	{"a":null, "b":-38341291.75793421}	500	200 o 400	Fallito
13	POST	{"a":"DiK8RLFg@D15a", "b":7882559.528860167}	500	200 o 400	Fallito
14	POST	{"a":-93811755.18476978, "b":true}	500	200 o 400	Fallito
15	POST	{"a":-68164064.24413502, "b":null}	500	200 o 400	Fallito
16	POST	{"a":-10676399.194627836, "b":"DS5*W"}	500	200 o 400	Fallito

Tabella 3.3. I risultati dei test relativi all'endpoint `divisione_ko`.

La Tab. 3.3 riporta invece i risultati e le informazioni relative ai test per l'endpoint `divisione_ko`.

Come si vede dalla Tab. 3.3 la maggior parte dei test eseguiti finiscono con il fallire. Gli unici ad andare a buon fine sono i numeri 1, 8, ovvero i test in cui le chiamate sono fatte in maniera corretta, e i numeri 2, 4, 5 e 7. I test che falliscono si possono poi dividere in due gruppi. Da un lato i test 3 e 6 falliscono a causa di una eccezione di tipo `nullPointerException` lanciata dal servizio. L'implementazione del metodo GET dell'endpoint `divisione_ko` come detto in precedenza non fa nessun controllo, quindi in questi due casi, in cui prima `a` è `null` e poi `b` il servizio ritorna 500. Dall'altro lato i test 9, 10, 11, 12, 13, 14, 15 e 16 falliscono sempre perché il servizio ha ritornato 500. In questi casi però il motivo è dovuto al fatto che il componente della libreria non è stato in grado, giustamente, di deserializzare il corpo JSON della richiesta e mapparlo su un oggetto di tipo `Numeri`. I fallimenti sono in questi casi dovuti a errori (qui voluti) nell'implementazione del servizio e nella gestione delle eccezioni.

Il secondo servizio usato per testare l'applicazione è un semplice servizio che rispetto al precedente può essere considerato più conforme ai principi che definiscono l'architettura REST. Si tratta di un servizio che permette di gestire un database di utenti. Tale servizio segue i principali vincoli dell'architettura REST, ovvero:

- gli endpoint esposti coincidono con le risorse del servizio;
- le operazioni sulle risorse sono mappate sui metodi HTTP;
- l'interazione client server avviene attraverso lo scambio delle rappresentazioni delle risorse.

Nella scala di maturità di Richardson, tale servizio, sebbene molto semplice, si assesta al livello 2.

Il servizio espone due endpoint:

- `/users`;
- `/users/userId`.

Il primo gestisce l'intera collezione di utenti. Il secondo invece dà accesso alla singola risorsa utente. La rappresentazione di un utente segue un semplice schema JSON riportato in Fig. 3.2.

```
User:
  type: object
  properties:
    id:
      type: integer
      minimum: 0
    name:
      type: string
    age:
      type: integer
      minimum: 0
  required:
    - id
    - name
    - age
```

Figura 3.2. Lo schema JSON della risorsa `user`.

La risorsa contiene tre proprietà tutte obbligatorie. Di queste due sono di tipo `integer` e l'altra è di tipo `string`. Sulla proprietà `name` non vi è alcun vincolo aggiuntivo, viceversa per i due interi è richiesto che siano entrambi positivi.

Il primo endpoint prevede due metodi: GET e POST. Il metodo GET restituisce un array contenente tutti gli utenti noti al servizio. Il metodo POST aggiunge un nuovo utente al servizio a patto che non ne esista uno con lo stesso id. In questo caso il servizio risponde con un codice di stato della risposta 409 - Conflict. Se il metodo POST restituisce un success (201 - Created) il servizio ritorna la risorsa appena creata e rimanda all'URL che individua la risorsa sul servizio.

Il secondo endpoint prevede tre metodi: GET, PUT e DELETE. Il metodo GET restituisce la rappresentazione dell'utente associato all'id `userId`, ovvero il parametro usato nel path. Il metodo PUT è usato per aggiornare la risorsa. Quando si invoca tale metodo il servizio si aspetta che il corpo contenga un JSON conforme allo schema relativo all'utente e che il campo `id` abbia lo stesso valore del parametro che compare nel path. Se così non fosse il servizio restituisce un errore 400 - bad request. L'ultimo metodo, DELETE, cancella la risorsa dal servizio se presente. In tutti i casi se l'id non individua alcuna risorsa sul servizio si ottiene la risposta 404 - Not found.

La Tab. 3.4 mostra i risultati dei test eseguiti sull'endpoint `/users`.

Tutti i test vanno a buon fine. Il primo test crea un nuovo utente sul servizio. L'id generato non corrisponde ad un utente già presente, il corpo della richiesta è ben formato e la risorsa viene creata con successo. In questo caso l'asserzione relativa al codice di stato della risposta è puntuale. I test 2, 3, e 4 sono i test che vengono generati a partire dai dataset proposti dal generatore. Tutti questi dataset condividono il fatto che mancano di una proprietà che nello schema JSON è dichiarata come obbligatoria. Nei dataset impiegati nei test 5, 6, 7, 8 e 9, invece la proprietà `id` ha un tipo errato. Tra tutti però il test numero 6 restituisce 201 sebbene al campo `id` sia stato assegnato un `number` e non un `integer`. Ciò è dovuto al fatto che il parser del framework usato per implementare il servizio forza la deserializzazione del corpo della richiesta e trasforma il valore 60.04 in 60. Negli altri test invece la risposta ottenuta è 400. Lo stesso accade nei test 10, 11, 12 e 13. Qui è la proprietà `name` a non essere valorizzata correttamente. Tuttavia, come visto nel caso precedente, la libreria che si occupa della deserializzazione trasforma il contenuto assegnato a `name`

# *Test*	*Metodo*	*Parametri*	*Risposta*	*Risposta attesa*	*Esito*
1	POST	{"id":69335538, "name":"fugiat", "age":36629857}	201	201	Superato
2	POST	{"name":"laborum enim", "age":54204302}	400	201 o 400 o 409	Superato
3	POST	{"id":8002892, "age":25002623}	400	201 o 400 o 409	Superato
4	POST	{"id":43202852, "name":"mollit consectetu"}	400	201 o 400 o 409	Superato
5	POST	{"id":true,"name":"Lorem", "age":37518073}	400	201 o 400 o 409	Superato
6	POST	{"id":60.04,"name":"eu est in non","age":22617824}	201	201 o 400 o 409	Superato
7	POST	{"id":null,"name":"Duis", "age":89727642}	400	201 o 400 o 409	Superato
8	POST	{"id":"vz*xxRXIzQ","name": "anim dolor nostrud","age":55025193}	400	201 o 400 o 409	Superato
9	POST	{"id":-1,"name":"sed adipisicing nostrud","age":26597424}	400	201 o 400 o 409	Superato
10	POST	{"id":93268219,"name":false, "age":79699625}	201	201 o 400 o 409	Superato
11	POST	{"id":89252060, "name":6528053484191744, "age":71829361}	201	201 o 400 o 409	Superato
12	POST	{"id":53395629,"name":78.02, "age":87869490}	201	201 o 400 o 409	Superato
13	POST	{"id":7838233,"name":null, "age":34518057}	400	201 o 400 o 409	Superato
14	POST	{"id":78346466,"name":"sed", "age":true}	400	201 o 400 o 409	Superato
15	POST	{"id":62370153, "name":"incididunt magna tempor dolore","age":46.28}	201	201 o 400 o 409	Superato
16	POST	{"id":91388032, "name":"eiusmod", "age":null}	400	201 o 400 o 409	Superato
17	POST	{"id":40207280,"name":"culpa consequat veni","age":"ap0IfibvDRG1K)W4"}	400	201 o 400 o 409	Superato
18	POST	{"id":11319994,"name":"tempor qui cupidatat","age":-1}	400	201 o 400 o 409	Superato

Tabella 3.4. I risultati dei test relativi all'endpoint `/users`.

in una stringa. Pertanto, nel test 10, per esempio, l'oggetto creato dal servizio quando giunge la richiesta ha l'attributo `name` valorizzato con "false". Infine, per i test 14, 15, 16, 17 e 18 valgono le stesse considerazioni fatte per i test 5, 6, 7, 8 e 9.

Se si riprova poi, dopo una prima esecuzione, ad eseguire nuovamente la test suite, tutti i test che in precedenza avevano ritornato 400 continueranno a ritornare 400. Viceversa i test in cui si era ottenuto 201 falliscono, perché la risorsa che ogni test sta cercando di creare è già presente, in quanto già aggiunta nella precedente run. Il servizio quindi ritornerà 409.

Per il secondo endpoint vengono generati in tutto 23 test. Di questi 2 sono relativi al metodo GET, 2 sono relativi al metodo DELETE e i restanti si riferiscono invece tutti al metodo PUT. Per generare i test si è prima proceduto a generare e compilare opportunamente il JSON contenente i

valori da assegnare al parametro `userId` per ogni terna prevista.

La Tab. 3.5 mostra i risultati dei test relativi ai metodi GET e DELETE eseguiti sull'endpoint `/users/userId`.

# Test	*Metodo*	*Parametri*	*Risposta*	*Risposta attesa*	*Esito*
1	GET	userId = 5	200	200	Superato
2	GET	userId = 8	404	404	Superato
3	DELETE	userId = 10	200	200	Superato
4	DELETE	userId = 11	404	404	Superato

Tabella 3.5. I risultati dei test relativi all'endpoint `/users/userId` per i metodi GET e DELETE.

Entrambi i test relativi al metodo GET vanno a buon fine. Il primo infatti accede a un utente che esiste sul servizio. Il secondo invece riceve 404 dal servizio in quanto non esiste alcun utente con id 8. Le stesse considerazioni valgono per i test relativi al metodo DELETE. In tutti e quattro i test al parametro `userId` sono stati assegnati i valori leggendoli dal file contenente i valori di tipo `path`.

La Tab. 3.6 mostra invece i risultati dei test relativi al metodo PUT.

Degli altri 19 test, 18 vanno a buon fine e uno fallisce. A fallire è il test numero 3. Il servizio infatti impone che affinché l'aggiornamento della risorsa vada a buon fine è necessario che il nuovo utente abbia come id il valore associato all'identificativo della risorsa sul servizio, ovvero il valore di `userId`. Questa condizione poteva certamente essere tralasciata in fase di implementazione del servizio. Tuttavia risulta una condizione plausibile e corretta al fine di mantenere coerenza e ordine tra i dati posseduti dal servizio e le corrispondenti rappresentazione esposte al client. Si tratta dunque di una condizione particolare imposta dal servizio che l'applicazione che genera i test non può capire autonomamente. Osservando i dataset usati per i test relativi al metodo PUT nessuno di questi ha come id lo stesso valore assegnato a `userId`, cioè 5. È per questo motivo che il test numero 3 fallisce. Gli altri 18 vanno a buon fine ma la risposta del servizio è sempre 400 e ciò è dovuto, in alcuni casi (9, 13, 14, 15 e 18) al controllo fatto dal servizio sull'uguaglianza degli id. D'altronde al parametro `userId` è stato assegnato un valore manualmente, mentre ai corpi delle richieste sono stati assegnati dei valori generati casualmente. In questi casi, per evitare situazioni del genere, l'alternativa è quella di ritoccare i test e modificare, nel caso specifico, il valore della proprietà `userId` nei body con lo stesso valore del parametro `path`.

In ultimo, si riportano i risultati ottenuti tramite un servizio preesistente e che, a differenza dei precedenti, non è stato implementato appositamente per testare l'applicazione. Si tratta di un servizio per la gestione di referti che costituisce uno dei servizi di back-end per una web application. Il servizio espone diversi endpoint, di questi però solo quattro sono stati testati con l'applicazione. Il servizio prevede il metodo di autenticazione basic auth. Tale schema di sicurezza è applicato a tutti gli endpoint del servizio. Il servizio prevede poi alcune regole che si applicano a tutti o solo ad alcuni endpoint. Se non si seguono tali regole quando si interroga il servizio quest'ultimo risponderà con un codice di errore. La prima regola riguarda l'uso di un parametro di tipo `header`, chiamato `X-HTTP-CODICEFISCALE`. La regola prevede che questo parametro venga valorizzato con un codice fiscale. La maggior parte degli endpoint include un parametro `path` che individua un utente sul servizio. Questi identificativi sono i codici fiscali degli utenti. La regola prevede che, per gli endpoint che includono questo parametro di tipo path, al parametro `header` si assegni lo stesso valore del parametro che individua l'utente. L'ultima regola prevede invece che per alcuni endpoint sia valorizzato anche un parametro di tipo `query` chiamato `pin`. Tale parametro va valorizzato con il codice pin associato all'utente. In caso di pin sbagliato il servizio restituisce errore.

Per gestire e tenere conto di ognuna di queste tre regole sono state sfruttate tre diverse caratteristiche dell'applicazione già esposte nel capitolo precedente. Per quanto riguarda la basic authentication non sono state necessarie ulteriori modifiche, in quanto il modulo swagger-test-templates tiene in considerazione automaticamente gli schemi di autenticazione. Ovviamente, la descrizione del servizio riportava il tag `security` al livello più esterno nella descrizione, in modo da applicare lo schema di sicurezza a tutti gli endpoint. L'impiego di tale tag provoca, in fase di

# Test	Metodo	Parametri	Risposta	Risposta attesa	Esito
5	PUT	userId = 5, {"id": 44373424, "name": "nulla sint sunt", "age": 83551063}	400	200	Fallito
6	PUT	userId = 90, {"id": 44373424, "name": "nulla sint sunt", "age": 83551063}	404	404	Superato
7	PUT	userId = 5, {"name":"esse","age":79933143}	400	200 o 400 o 404	Superato
8	PUT	userId = 5, {"id":7083405,"age":42049499}	400	200 o 400 o 404	Superato
9	PUT	userId = 5, {"id":14853604,"name":"anima"}	400	200 o 400 o 404	Superato
10	PUT	userId = 5, {"id": true,"name": "qui tempor","age": 13915226}	400	200 o 400 o 404	Superato
11	PUT	userId = 5, {"id": 88.84,"name": "voluptate velit dolor","age": 30894639}	400	200 o 400 o 404	Superato
12	PUT	userId = 5, {"id": null,"name": "cupidatat sunt nisi et","age": 26203820}	400	200 o 400 o 404	Superato
13	PUT	userId = 5, {"id": "Ce0D9z","name": "ipsum","age" : 23511035}	400	200 o 400 o 404	Superato
14	PUT	userId = 5, {"id": -1,"name": "in qui","age": 21614572}	400	200 o 400 o 404	Superato
15	PUT	userId = 5, {"id": 19939791,"name": true, "age": 43820351}	400	200 o 400 o 404	Superato
16	PUT	userId = 5, {"id": 10110824,"name": -1355376, "age": 932}	400	200 o 400 o 404	Superato
17	PUT	userId = 5, {"id": 39935689,"name": 10.8, "age": 38029330}	400	200 o 400 o 404	Superato
18	PUT	userId = 5, {"id": 24072146,"name": null, "age": 47879837}	400	200 o 400 o 404	Superato
19	PUT	userId = 5, {"id": 67718639,"name": "commodo", "age": true}	400	200 o 400 o 404	Superato
20	PUT	userId = 5, {"id": 68743746,"name": "aute elit ullamco","age": 99.19}	400	200 o 400 o 404	Superato
21	PUT	userId = 5, {"id": 99089891,"name": "anima", "age": null}	400	200 o 400 o 404	Superato
22	PUT	userId = 5, {"id": 40147592, "name": "lorem ipsum", "age": "R9fxz*pQ7v"}	400	200 o 400 o 404	Superato
23	PUT	userId = 5, {"id": 33491234, "name": "laborum", "age": -1}	400	200 o 400 o 404	Superato

Tabella 3.6. I risultati dei test relativi all'endpoint `/users/userId` per il metodo PUT.

generazione dei test, l'aggiunta del parametro `Authorization`, valorizzato con la stringa 'Basic ' seguita dal token letto dal file `.env`.

Per la gestione corretta del pin si è fatto affidamento al tag personalizzato `x-example`, che è stato aggiunto all'interno della dichiarazione del parametro `pin` in tutti i punti della descrizione in cui veniva usato. Al tag è stato assegnato il valore del pin corretto. In fase di generazione di test, l'applicazione utilizza il valore di `x-example` per quelle chiamate in cui il parametro `pin` deve essere valorizzato correttamente, mentre, per le chiamate in cui è `pin` ad assumere un valore non corretto si impiega il generatore.

Infine, per gestire i valori da assegnare in ogni chiamata al parametro X-HTTP-CODICEFISCALE si è utilizzato il file JSON in cui l'utente può assegnare manualmente dei valori a certi parametri. In fase di generazione del file JSON si è specificato il parametro header come parametro che deve comparire nello scheletro. Poi si è provveduto a compilare lo scheletro in maniera coerente.

Tutti gli endpoint prevedono il solo metodo GET. Il primo endpoint che è stato testato non prevede altri parametri in ingresso oltre al parametro `X-HTTP-CODICEFISCALE`. In questo caso visto che l'endpoint non prevede il parametro `path` che individua l'utente a `X-HTTP-CODICEFISCALE` può essere assegnato qualunque valore. Per questo endpoint vengono generati due test ed entrambi vanno a buon fine.

Il secondo endpoint testato consente di scaricare il pdf dei referti. L'endpoint prevede:

- un primo parametro path che individua l'utente;
- un secondo parametro path che individua il referto;
- il pin;
- una stringa che consente al chiamante di richiedere un documento firmato o meno.

In totale vengono generati 18 test. Di questi:

- 4 prevedono il codice 200 in quanto sono richieste ben formate;
- 4 prevedono il codice 404 in quanto si usano parametri path a cui non corrisponde una risorsa;
- i restanti sono relativi a un errore nel valorizzare il parametro pin.

Eseguendo i test però tutti i test falliscono in quanto dal servizio è arrivato un codice di stato della risposta non dichiarato nella descrizione.

Per gli altri due endpoint testati si sono verificati invece dei comportamenti simili. Il primo endpoint prevede in ingresso un parametro di tipo path che individua l'utente e un filtro, ovvero una stringa. Tale endpoint restituisce l'elenco di tutti i referti di uno specifico utente. Per questo endpoint vengono generati 8 test, di cui:

- 2 prevedono il codice 200 in quanto sono richieste ben formate;
- 2 prevedono il codice 404 in quanto si usano parametri path a cui non corrisponde una risorsa;
- i restanti sono relativi a un errore nel valorizzare il parametro relativo al filtro.

Eseguendo i test una prima volta, tutti tranne quelli che si aspettano 200 vanno a buon fine. A fallire non è però l'asserzione relativa al codice di stato della risposta, bensì l'asserzione relativa al corpo della risposta. Da una successiva analisi è emersa una discordanza nei nomi che si erano assegnati ad alcuni formati come per esempio quello che viene usato per esprime la data e l'ora. Lo schema nella descrizione riportava come nome `date-time` mentre il validatore della libreria ZSchema era registrato sotto il nome `dateTime`. Riallineando i nomi anche i primi 2 test vanno a buon fine.

L'ultimo endpoint a differenza del precedente prende in più come parametri l'id del referto come parametro path e il pin. Per questo endpoint i test generati sono 20, di cui:

- 2 prevedono il codice 200 in quanto sono richieste ben formate;
- 4 prevedono il codice 404 in quanto si usano parametri path a cui non corrisponde una risorsa;
- i restanti sono relativi a un errore nel valorizzare il parametro relativo al filtro o del pin.

Anche in questo caso il risultato dell'esecuzione dei test rispecchia la situazione appena esposta per il precendente enpoint.

Capitolo 4

Conclusioni

Considerando il punto di partenza e gli strumenti a disposizione, il risultato finale può considerarsi complessivamente soddisfacente. Ovviamente l'applicazione nel suo stato attuale ha molti aspetti che possono essere migliorati, ma comunque al momento soddisfa il requisito iniziale di strumento in grado di testare automaticamente un servizio web.

Certamente, l'aspetto più vantaggioso nell'uso dell'applicazione è la sua capacità di generare automaticamente unità di test che altrimenti andrebbero scritte manualmente. Tutto questo è possibile grazie all'uso di JavaScript e ai template Mustache che consentono insieme di creare dei file .js che sono a tutti gli effetti degli script autogenerati ed eseguibili simili a quelli che un programmatore scriverebbe a mano. Inoltre JavaScript risulta di estrema comodità nell'ottica della comprensione naturale del formato JSON, largamente usato nei servizi REST. Inoltre il punto di partenza, ovvero l'unico prerequisito dell'applicazione, è la presenza di una descrizione del servizio. Si tratta comunque di un prerequisito semplice da soddisfare, soprattutto se si pensa che la descrizione di un servizio torna molto utile nell'interazione tra diversi gruppi di lavoro e in generale quando chi usa il servizio non coincide con chi lo ha implementato. Il punto di forza è sicuramente la capacità di individuare grossi errori in fase di implementazione, come ad esempio errori di 500 - Internal server error, dovuti per esempio a null pointer exception.

Ma come detto ci sono diversi aspetti che potrebbero essere migliorati. Il primo aspetto che si potrebbe migliorare è supportare la versione 3.0 di Swagger, ribattezzata OpenAPI 3. Attualmente l'applicazione è in grado di comprendere descrizioni che usano lo schema definito dalla seconda versione. Rispetto alla versione 2.0, la 3.0 introduce dei nuovi elementi, inoltre i nomi e la posizione di alcuni tag sono cambiati. Rimane però sempre possibile, nel caso si voglia usare l'applicazione ma si dispone solo di una descrizione conforme alla terza versione convertire la descrizione usando dei tool online oppure riscrivere la descrizione seguendo le regole imposte dalla seconda versione.

L'altro aspetto che si potrebbe migliorare è il carattere generale che in certe situazione assume l'asserzione relativa al codice di stato della risposta. Come si è visto in precedenza infatti se la descrizione è poco dettagliata l'applicazione genera alcuni test in cui l'asserzione relativa al codice di stato della risposta non è puntuale, bensì verifica che la risposta sia una tra quelle elencate nella descrizione.

Un altro aspetto su cui poter ancora lavorare è il generatore. Anche in questo caso infatti se la descrizione offre poche informazioni, per esempio il formato di un parametro di tipo String o l'intervallo di valori che un parametro di tipo intero può assumere, il generatore tende a generare dei valori che sono molto diversi da quelli che normalmente un utente genererebbe. La soluzione in questo caso potrebbe essere quella di metter mano al generatore, implementando nuovi comportamenti, oppure sostituirlo con un nuovo mantenendo inalterata l'interfaccia di comunicazione con il resto dell'applicazione.

Infine, come si è visto nel capitolo dedicato ai risultati, nel caso del servizio degli utenti, l'applicazione non è in grado per sua natura di cogliere degli aspetti dell'implementazione del servizio che sono noti a chi lo implementa e che non possono essere espressi nella descrizione. Per questo tipo di situazione attualmente l'unica alternativa consiste come detto nel ritoccare manualmente i

test aggiungendo possibilmente delle ulteriori asserzioni personalizzate oppure modificando i dati impiegati per compiere le chiamate HTTP al servizio.

Bibliografia

[1] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard, "Web Services Architecture", W3C Working Group Note, 11 February 2004, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[2] Understanding HATEOAS, http://spring.io/understanding/HATEOAS

[3] REST APIs must be hypertext-driven, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[4] L. Richardson, "Justice Will Take Us Millions Of Intricate Moves", "Act Three: The Maturity Heuristic", QCon San Francisco 2008 Conference, November 2008, https://www.crummy.com/writing/speaking/2008-QCon/act3.html

[5] M. Hadley, "Web Application Description Language", http://www.w3.org/Submission/wadl/

[6] OpenAPI github project, http://github.com/OAI/OpenAPI-Specification

[7] SWAGGERhub platform web page, http://swaggerhub.com/

[8] RAML website, https://raml.org/

[9] API Blueprint github project, https://github.com/apiaryio/api-blueprint

[10] Dredd documentation, https://dredd.readthedocs.io/en/latest/how-it-works.html#making-your-api-description-ready-for-testing

[11] Node.js website, https://nodejs.org/en/

[12] Mocha website, https://mochajs.org/

[13] Jersey github project, https://jersey.github.io/

[14] JAX-RS docs, https://docs.oracle.com/javaee/6/tutorial/doc/giepu.html