



UNIVERSITÀ DEGLI STUDI DI CATANIA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

MARIO BENISSIMO

COOKER: UN FRAMEWORK PER LO SVILUPPO
AUTOMATICO DI API TEST

ELABORATO FINALE

prof. Emiliano Tramontana
prof. Andrea Francesco Fornaia

Anno Accademico 2023–2024

Indice

1	Evoluzione dei Paradigmi API: Dalla Flessibilità di REST a GraphQL e gRPC	2
2	La rinascita della scrittura dei Test: L’Influenza dei processi CI/CD nell’ecosistema dello sviluppo Software	5
3	Ottimizzazione dei Costi attraverso l’Automation Testing	6
4	API Testing: Un’analisi delle diverse metodologie	7
5	Cooker	10
5.1	Premesse	10
5.2	Cooker Test: Fuzz Test, Security Test e Stress Testing	10
5.3	Descrizione degli Endpoint: Guida per la creazione dei Test con Cooker	11
5.4	Semplificare la creazione dei JSON tramite interfaccia grafica	13
5.5	Dettagli tecnici	14
6	Fattori esterni e indipendenti dal framework	17
7	Architettura	21
8	Esempi di utilizzo	22
8.1	Introduzione	22
8.2	Esempio 1	22
8.3	Esempio 2	23
8.4	Esempio 3	24
9	Continuous Integration and Continuous Delivery (CI/CD)	26
10	Futuri Sviluppi	29
11	Conclusioni	30

Sommario

Cooker nasce come il compagno ideale dei sviluppatori, offrendo una soluzione all'avanguardia per velocizzare la creazione di test API. Attraverso una descrizione informale degli endpoint, Cooker genera una serie di test mirati a individuare imperfezioni e promuovere la coerenza nella corretta implementazione delle interfacce API. Una risorsa imprescindibile per perfezionare e semplificare il processo di testing.

1 Evoluzione dei Paradigmi API: Dalla Flessibilità di REST a GraphQL e gRPC

Prima di immergerci nei dettagli tecnici del framework, concediamoci una breve incursione nel panorama attuale della scrittura delle API. Oggi, sentiamo spesso parlare di **API RESTful**, un termine che trae origine da 'REST' (Representational State Transfer), delineando un approccio al trasferimento di stato dei dati. Va sottolineato che REST non rappresenta una tecnologia in sé, bensì si configura come un insieme di linee guida e approcci che definiscono lo stile di trasmissione dei dati. Di conseguenza, un insieme di API che segue fedelmente la logica di REST dà vita a un set di API 'RESTful'. Tuttavia, definire un'architettura standard per le API REST risulta un'impresa complessa. Questa mancanza di rigidità offre agli sviluppatori una libertà maggiore nella definizione degli endpoint, ma allo stesso tempo impedisce di stabilire una struttura uniforme che tra l'altro faciliterebbe anche la scrittura di test automatici. Nonostante questa flessibilità, esistono delle linee guida generali, comunemente presenti in molte applicazioni. Un esempio sono gli `http status`, i quali agevolano e accelerano le risposte dai server, costituendo un fondamentale punto di riferimento nell'ecosistema delle API.

Nonostante si interagisce con delle API, normalmente siamo impegnati in operazioni con un database ospitato su un server online. Le principali operazioni che possono essere svolte sono descritti dai metodi di richiesta:

- Il metodo **GET** ci consente di richiedere al server un determinato insieme di dati;
- Con il metodo **POST** otteniamo la possibilità di creare un nuovo oggetto all'interno del database;

- Mediante il metodo **PUT**, siamo in grado di modificare o rimpiazzare integralmente un oggetto già esistente;
- Con il metodo **DELETE**, abbiamo la facoltà di cancellare da remoto un oggetto contenuto nel database al quale siamo connessi.

Questi fondamentali metodi delineano le operazioni di base che, orchestrando le interazioni con il database.

I metodi di richiesta, gli HTTP status code e altre linee guida sono elementi fondamentali nella scrittura degli endpoint. Nell'immagine sottostante, possiamo vedere alcune linee guida che molto spesso vengono utilizzate come struttura base per la scrittura di endpoint.



Design a Shopping Cart

Use resource names (nouns)	✗ GET /querycarts/123	✓ GET /carts/123
Use plurals	✗ GET /cart/123	✓ GET /carts/123
Idempotency	✗ POST /carts	✓ POST /carts {requestId: 4321}
Use versioning	✗ GET /carts/v1/123	✓ GET /v1/carts/123
Query after soft deletion	✗ GET /carts	✓ GET /carts? includeDeleted=true
Pagination	✗ GET /carts	✓ GET /carts? pageSize=xx&pageToken=xx
Sorting	✗ GET /items	✓ GET /items? sort_by=time
Filtering	✗ GET /items	✓ GET /items? filter=color:red
Secure Access	✗ X-API-KEY=xxx	✓ X-API-KEY = xxx X-EXPIRY = xxx X-REQUEST-SIGNATURE = xxx <i>hmac(URL + QueryString + Expiry + Body)</i>
Resource cross reference	✗ GET /carts/123? item=321	✓ GET /carts/123/items/321
Add an item to a cart	✗ POST /carts/123? addItem=321	✓ POST /carts/123/items:add { itemId: "items/321" }
Rate limit	✗ No rate limit - DDos	✓ Design rate limiting rules based on IP, user, action group etc

Figura 1: Best practices to design API [4]

Nel panorama attuale, lo stato dell'arte introduce anche il concetto di GraphQL, un'alternativa che porta con sé una flessibilità superiore rispetto a REST. Tale approccio consente alle applicazioni di richiedere esclusivamente i dati necessari, offrendo un controllo più preciso sul flusso informativo. GraphQL sta guadagnando notevole popolarità, soprattutto nelle applicazioni che gestiscono dati complessi. Inoltre, emerge negli ultimi anni gRPC,

un moderno framework RPC (Remote Procedure Call) open source caratterizzato da prestazioni elevate. La sua versatilità gli consente di operare in qualsiasi ambiente e trova un ampio impiego nella comunicazione tra micro-servizi interni.

Questa recente evoluzione sottolinea l'incessante dinamismo nel mondo delle tecnologie, offrendo soluzioni sempre più avanzate e adattabili alle esigenze delle moderne architetture.

2 La rinascita della scrittura dei Test: L'Influenza dei processi CI/CD nell'ecosistema dello sviluppo Software

L'importanza del software testing risiede nel suo ruolo cruciale di garantire la qualità e l'affidabilità di un'applicazione software. Un processo di testing completo non solo individua e corregge errori e difetti nel codice, ma contribuisce a fornire un prodotto finale più stabile e sicuro per gli utenti.

Negli ultimi anni, l'adozione diffusa delle pratiche di **Continuous Integration (CI)** e **Continuous Delivery (CD)** ha notevolmente influenzato l'incremento della scrittura di test nel processo di sviluppo del software. L'integrazione continua implica la costante fusione del codice da parte degli sviluppatori, seguita da una serie di test automatici per assicurare che le nuove modifiche non compromettano l'integrità del sistema esistente. Parallelamente, la Continuous Delivery si concentra sull'automazione del rilascio del software in ambienti di produzione.

Le CI/CD, accelerano il ciclo di sviluppo, riducono i rischi di implementazione e migliorano la qualità complessiva del software. Tuttavia, per

garantire l'efficacia delle CI/CD, è essenziale avere un robusto insieme di test automatizzati che coprano ampiamente le funzionalità dell'applicazione. Di conseguenza il testing non è solo un passo finale nel ciclo di sviluppo del software, ma è integrato in tutto il processo, permettendo una verifica continua della corretta implementazione delle funzionalità e delle prestazioni del sistema.

3 Ottimizzazione dei Costi attraverso l'Automation Testing

I costi complessivi associati ai test del software oscillano tra il 15% e il 25% del costo totale di un progetto, seguendo gli standard del settore. Un rapporto emesso nel 2019 da un gruppo di CIO e senior engineer ha rivelato che, in quell'anno, circa il 23% del budget IT annuale delle loro organizzazioni è stato allocato, in media, per attività di controllo della qualità e test. Il numero di casi da testare, in parallelo all'espansione di un progetto, può crescere in maniera esponenziale, rendendo i costi spesso insostenibili.

Una distinzione fondamentale nei test si articola tra quelli manuali e quelli automatici. I test manuali, creati manualmente dai tester, richiedono un investimento temporale maggiore, a differenza dei test automatici, generati tramite strumenti sofisticati o script. Tale distinzione è chiaramente delineata nella figura sottostante. Nell'ambito aziendale, il 92% delle organizzazioni, indipendentemente dalle dimensioni, abbraccia una forma di automazione attraverso l'utilizzo di strumenti appositi, tentando di abbattere i costi economici e temporali legati alla scrittura dei test.

Nonostante i notevoli vantaggi ottenuti attraverso una pratica diligente di test automatizzati, è fondamentale sottolineare che ciò non implica la

completa sostituzione dei test manuali. Un approccio ottimale al processo di test del software prevede sempre una sinergia tra casi di test manuali e automatizzati, garantendo così elevati standard di qualità, efficienza e un utilizzo ottimale delle risorse disponibili.

Aspetto del test	Manuale	Automazione
Esecuzione del test	Fatto manualmente dai tester del QA	Fatto automaticamente utilizzando strumenti e script di automazione
Efficienza del test	Richiede tempo e meno efficiente	Più prove in meno tempo e maggiore efficienza
Tipi di attività	Attività interamente manuali	La maggior parte delle attività può essere automatizzata, comprese le simulazioni di utenti reali
Copertura del test	Difficile garantire una copertura di prova sufficiente	Facile da garantire una maggiore copertura del test

Figura 2: Test Manuale vs Test Automation

4 API Testing: Un'analisi delle diverse metodologie

In questo paragrafo, esploriamo le principali tipologie di testing, focalizzandoci in modo specifico su come possano essere adattate in modo esclusivo per testare le API. Tra le fondamentali metodologie di verifica, emergono:

- **Smoke Testing:** Questa pratica si limita a verificare se l'endpoint funziona e se è raggiungibile, senza approfondire la logica di interrogazione e risposta.

- **Test Funzionali:** Questi test si concentrano sull'assicurare che l'endpoint restituisca il risultato atteso, valutando la correttezza delle sue funzionalità.
- **Integration Testing:** Tale approccio verifica l'interazione tra diversi endpoint, garantendo una coesione fluida tra le diverse componenti.
- **Regression Testing:** Mira a verificare che non si siano verificati problemi negli endpoint esistenti durante le correzioni o modifiche di altri endpoint.
- **Load Testing:** Questa pratica valuta la capacità dell'applicazione simulando un normale flusso di traffico in ingresso.
- **Stress Testing:** Concentrandosi su un carico di traffico elevato, questo tipo di test mette alla prova la robustezza dell'applicazione sotto pressione.
- **Security Testing:** Includono test di penetrazione simili a un attacco reale, questo tipo di verifica non solo valuta il rischio ma identifica e analizza le principali minacce sulla sicurezza.
- **UI Testing:** Questa metodologia verifica le interazioni tra l'interfaccia utente (UI) e gli endpoint, garantendo una coerenza efficace.
- **Fuzz Testing:** Identifica possibili vulnerabilità, questo test invia dati non corretti all'interno delle API, esplorando le potenziali debolezze del sistema.

È possibile avere una visione unificata di queste metodologie nell'immagine sottostante.

9 Types API Testing

blog.bytebytego.com

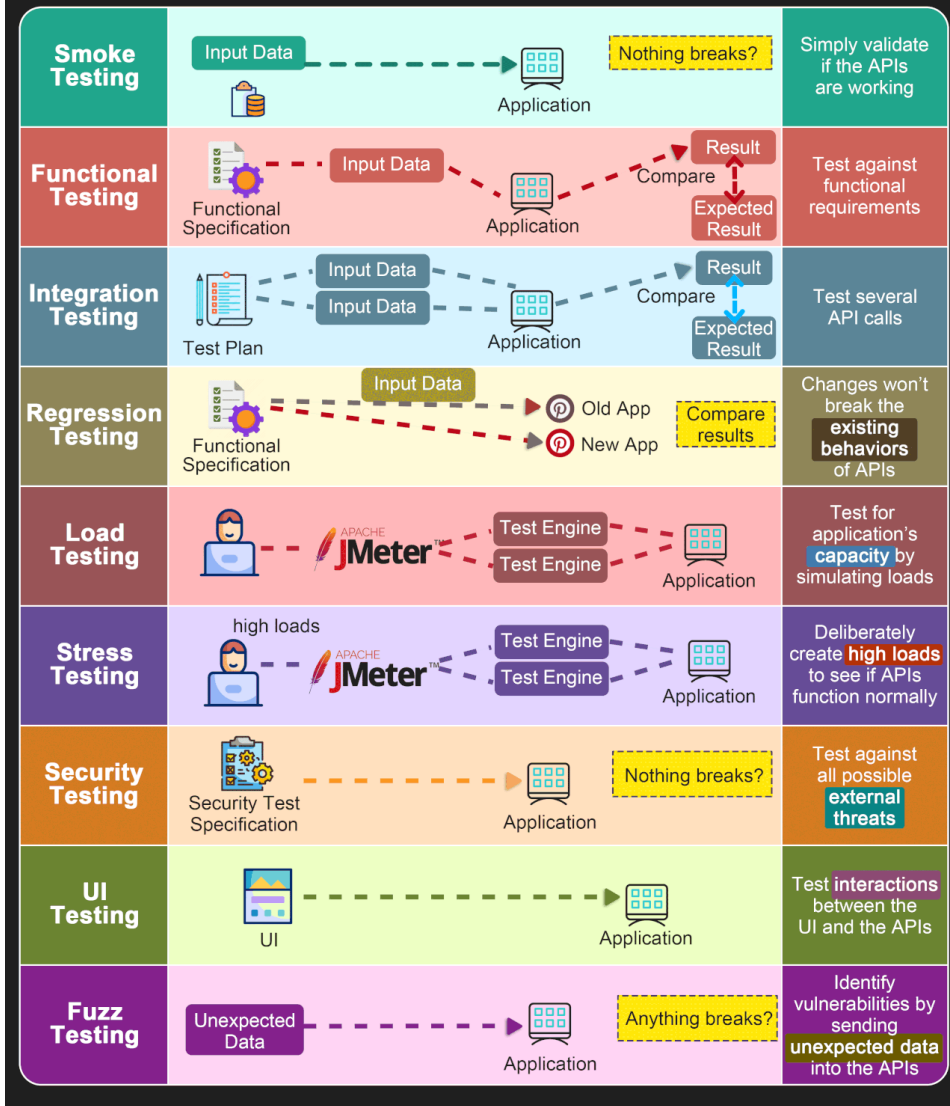


Figura 3: API testing [4]

5 Cooker

5.1 Premesse

Cooker si propone come un ambizioso strumento volto a generare automaticamente una suite di test, mirata a verificare la correttezza degli endpoint. Attualmente, il framework si presenta ancora in uno stato embrionale, dotato solamente delle funzionalità essenziali, con la chiara intenzione di espandere il numero di test case nelle versioni successive. La sfida principale che Cooker affronta risiede nella mancanza di uno standard ufficiale per la scrittura degli endpoint, costringendo il framework a fare delle assunzioni sulla correttezza di tali elementi. Le dettagliate spiegazioni di queste assunzioni saranno presentate nei paragrafi successivi. Cooker propone una suite di test generica, permettendo così la sua modifica da parte degli sviluppatori e adattamento a varie logiche di sviluppo.

5.2 Cooker Test: Fuzz Test, Security Test e Stress Testing

Nel processo di costruzione degli endpoint, dovrebbe essere instaurata una rigida definizione dei parametri accettati. Nonostante ciò, paradossalmente, la maggior parte delle vulnerabilità nasce da dati non corretti che si infiltrano all'interno dell'applicazione. Per affrontare questa problematica, Cooker implementa dei Fuzz Test, una metodologia volta a individuare combinazioni di parametri che possono causare malfunzionamenti. Tale approccio si basa su assunzioni che verificano se il valore restituito corrisponde alle aspettative. La stesura manuale di Fuzz Test risulta estremamente laboriosa, specie all'aumentare di parametri ed endpoint, con un conseguente incremento esponenziale del numero di test. Oltre alla creazione di Fuzz Test, Cooker genera test per verificare l'implementazione corretta del processo

di autenticazione, inclusi nei Security Test. Infine, il framework include un test dedicato a valutare il corretto funzionamento di un eventuale limitatore lato server, se implementato, rientrante nella tipologia di Stress Testing.

5.3 Descrizione degli Endpoint: Guida per la creazione dei Test con Cooker

Per consentire a Cooker di generare i test necessari, è essenziale fornire una descrizione per ciascun endpoint sotto forma di JSON. Il framework mette a disposizione una dashboard per agevolare la creazione di tali descrizioni, ma in alternativa, è possibile scriverle direttamente e posizionarle all'interno di una specifica cartella, dalla quale Cooker attingerà per generare i test.

Approfondiamo ora il corretto procedimento per creare un JSON che descrive un endpoint. Le chiavi obbligatorie sono **'endpoint'** e **'method'**, entrambe stringhe che forniscono il setup di base. Successivamente, è possibile specificare una lista di parametri, per ognuno dei quali è richiesto un **'name'** (nome del parametro), **'type'** (tipo del parametro) e un **'correctValue'** che rappresenta un valore di esempio accettato dall'endpoint. Per ulteriori dettagli e per identificare potenziali vulnerabilità, possono essere aggiunte chiavi aggiuntive le quali variano in base al tipo del parametro. Ad esempio, per i parametri di tipo **'string'**, è possibile specificare **'maxLength'**, indicando la massima dimensione della stringa accettabile dall'endpoint, mentre per i parametri di tipo **'int'**, può essere fornita la chiave **'range'**, permettendo di testare i limiti di upperbound e lowerbound.

Una prima assunzione riguarda il metodo GET, nel caso di una corretta interrogazione, ci aspettiamo valori di ritorno sotto forma di JSON. Pertanto, può essere specificata la chiave **'expectationLength'**, indicante il numero atteso di valori restituiti.

In caso di una richiesta corretta ad un endpoint, ci aspettiamo un HTTP status 200 - OK (201 nel caso di un POST con un nuovo elemento creato); in tutti gli altri scenari, dovremmo ottenere un HTTP status 400 - Bad request.

Se un endpoint richiede una fase preliminare di autenticazione, la descrizione deve includere la chiave **'authentication'**, specificando il **'method'** e l'eventuale **'secret'**. Attualmente, è possibile specificare solo 'JWT' come metodo di autenticazione. In questo caso, se un endpoint è interrogato senza autenticazione, ci aspettiamo un HTTP status 401 - Unauthorized.

Se lato server viene utilizzato un rate limiter, è possibile specificarlo tramite la chiave **'limiter'**. Esso richiede altri due parametri, rispettivamente **'maxRequests'** e **'seconds'**. Il primo indica il numero massimo di richieste consentite entro un determinato numero di secondi, specificato nel secondo parametro.

Il test generato effettua **'maxRequests'** richieste entro il numero di secondi indicato. Ci si aspetta che queste richieste vadano a buon fine. Entro il tempo specificato da **'seconds'**, viene effettuata un'ulteriore richiesta e, in questo caso, ci si aspetta che il limitatore lato server impedisca di gestire tale richiesta, restituendo il codice 429 - Too Many Requests. Infine, il test aspetta che il limitatore si resetta e invia una nuova richiesta, questa volta ci si aspetta che venga gestita correttamente e restituisca uno status HTTP 200 - OK.

Un riassunto completo delle regole di descrizione può essere visualizzato nell'immagine sottostante:

Key	Type	Accepted only				Mandatory	
endpoint	string	/				yes	
method	string	get - post				yes	
parameters	array	Key	Type	Accepted only	Mandatory	no	
		name	string	/	yes		
		type	string	int - string - uuid	yes		
		correctValue	depend on parameter	int - string - uuid	yes		
		range	string	formatted: "1-100" e.g.	yes if type is int		
		maxLength	int	/	yes if type is string		
authentication	string	Key	Type	Accepted only	Mandatory	no	
		method	string	JWT	yes		
		secret	string	/	yes if method is JWT		
limiter	object	Key		Type	Accepted only	Mandatory	no
		maxRequests		int	/	yes	
		seconds		int	/	yes	

Figura 4: Regole di descrizione degli endpoint

5.4 Semplificare la creazione dei JSON tramite interfaccia grafica

L'interfaccia mette a disposizione gli strumenti necessari per semplificare il processo di creazione e verifica delle descrizioni JSON. Essa si presenta come segue:

Insert your JSON here:

```

{
  "endpoint": "http://localhost:8080/user",
  "method": "get",
  "parameters": [
    {
      "name": "id",
      "type": "uuid",
      "correctValue": "550e8400-e29b-41d4-a716-446655448000"
    }
  ],
  "expectationLength": 1,
  "limiter": {
    "maxRequests": 10,
    "seconds": 1
  }
}

```

Result:

```

{
  "endpoint": "http://localhost:8080/user",
  "method": "get",
  "parameters": [
    {
      "name": "id",
      "type": "uuid",
      "correctValue": "550e8400-e29b-41d4-a716-446655448000"
    }
  ],
  "expectationLength": 1,
  "limiter": {
    "maxRequests": 10,
    "seconds": 1
  }
}

```

Save JSON

New JSON

Get Dummy JSON

Send JSON

Rules:

Key	Type	Accepted only	Mandatory																								
endpoint	string	/	yes																								
method	string	get - post	yes																								
parameters	array	<table> <thead> <tr> <th>Key</th> <th>Type</th> <th>Accepted only</th> <th>Mandatory</th> </tr> </thead> <tbody> <tr> <td>name</td> <td>string</td> <td>/</td> <td>yes</td> </tr> <tr> <td>type</td> <td>string</td> <td>int - string - uuid</td> <td>yes</td> </tr> <tr> <td>correctValue</td> <td>depend on parameter</td> <td>int - string - uuid</td> <td>yes</td> </tr> <tr> <td>range</td> <td>string</td> <td>formatted: '1-100' e.g.</td> <td>yes if type is int</td> </tr> <tr> <td>maxLength</td> <td>int</td> <td>/</td> <td>yes if type is string</td> </tr> </tbody> </table>	Key	Type	Accepted only	Mandatory	name	string	/	yes	type	string	int - string - uuid	yes	correctValue	depend on parameter	int - string - uuid	yes	range	string	formatted: '1-100' e.g.	yes if type is int	maxLength	int	/	yes if type is string	no
Key	Type	Accepted only	Mandatory																								
name	string	/	yes																								
type	string	int - string - uuid	yes																								
correctValue	depend on parameter	int - string - uuid	yes																								
range	string	formatted: '1-100' e.g.	yes if type is int																								
maxLength	int	/	yes if type is string																								
authentication	string	method	string	JWT	yes	no																					
		secret	string	/	yes if method is JWT																						
limiter	object	maxRequests	int	/	yes	no																					
		seconds	int	/	yes																						

For more details try dummy json for json example or view github repo: <https://github.com/murichenisimo/Cooker>

Info

This endpoint is configured to accept the GET method and requires the specification of certain essential parameters. It is necessary to provide mandatory fields such as 'name', 'type', and 'correctValue', along with the 'endpoint' and 'method' fields. In the case of a GET request, it is crucial to include the 'expectationLength' parameter, representing the expected number of elements returned in the response body. Optionally, the 'limiter' attribute is available if you wish to conduct a functional test based on the rate limiter.

Saved JSON:

JSON Salvato 1

Figura 5: Dashbaord

A sinistra, è ospitato un mini editor che consente la redazione delle descrizioni in formato JSON. Qui, è possibile scrivere le specifiche per gli endpoint, e nella parte inferiore, eventuali errori di natura sia sintattica che semantica verranno segnalati, basandosi sulle regole precedentemente discusse. Sono disponibili pulsanti dedicati per salvare un JSON, crearne uno nuovo e ottenere esempi di JSON, agevolando così la stesura di nuove descrizioni.

Una volta completata la creazione delle descrizioni, è possibile inoltrarle al backend, il quale darà avvio alla fase di generazione automatica del codice. Per offrire un panorama completo, l'interfaccia include anche la tabella dei vincoli/regole, come mostrato nella Figura 4.

5.5 Dettagli tecnici

L'intero framework è stato implementato utilizzando Golang, un linguaggio di programmazione open source sviluppato da Google. Il codice sorgente è liberamente accessibile nella repository GitHub:

<https://github.com/mariobenissimo/Cooker>

Di conseguenza, è possibile eseguire un clone della repository per sperimentare direttamente il framework. In alternativa, per integrare Cooker nei propri progetti, è sufficiente importarlo mediante la seguente dichiarazione da terminale:

```
go get github.com/mariobenissimo/Cooker
```

Una volta effettuato l'import, è possibile adottare le due modalità di utilizzo distinte. La prima offre la possibilità di interagire con un'interfaccia grafica e si esegue mediante il seguente comando:

```
cooker := Cooker.CreateCooker(port: ":8082")
```

Esso accetta un singolo parametro, rappresentato dalla porta sulla quale l'interfaccia sarà esposta. In assenza di specifiche, il sistema adotterà di default la porta 8088.

La seconda modalità di utilizzo, senza interfaccia grafica, consente la generazione dei test attingendo i file JSON da un percorso specificato come parametro:

```
Cooker.cook(path: "../cooker")
```

Entrambe le modalità offrono la flessibilità necessaria per adattare Cooker alle diverse esigenze del progetto.

Per la generazione automatica del codice sorgente, il framework ha fatto uso del concetto di AST, acronimo di **'Abstract Syntax Tree'** (Albero di Sintassi Astratta). Un AST costituisce una rappresentazione strutturata e gerarchica di un programma sorgente, in cui ogni nodo nell'albero rappresenta un'entità grammaticale del linguaggio di programmazione. Que-

sto approccio consente la creazione di un file sorgente costruendo un albero sintattico e utilizzando nodi specifici per la costruzione dei test.

Nel contesto del linguaggio di programmazione Go, l'AST è impiegato internamente dal compilatore per rappresentare il programma sorgente durante le fasi di parsing e analisi sintattica. L'AST può essere esplorato e modificato attraverso il pacchetto `'go/ast'` di Go, che mette a disposizione un'API per navigare e manipolare l'Abstract Syntax Tree in modo efficiente.

L'unica libreria esterna presente nei file di test, al di fuori delle librerie native di Golang, è `'github.com/stretchr/testify/assert'`, un pacchetto di assert. Si tratta di una libreria esterna che offre una serie di funzioni per il testing assertivo, semplificando la scrittura di test chiari e informativi all'interno del codice Go. Il pacchetto `'testify/assert'` viene comunemente utilizzato in combinazione con il framework di testing standard di Go, incluso nel pacchetto `'testing'`. La sua inclusione facilita la creazione di test e fornisce un insieme di funzionalità avanzate rispetto agli assert di base offerti dal pacchetto di testing. Per aggiungere questa libreria al proprio progetto, è sufficiente eseguire il seguente comando:

```
go get "github.com/stretchr/testify/assert"
```

Infine, per eseguire l'intera suite di test presente nel progetto Go, è sufficiente utilizzare il comando:

```
go test
```

Lanciato dalla radice del progetto, questo comando automaticamente individuerà tutti i file di test che seguono il convenzionale pattern di denominazione di golang, ovvero tutti quelli che terminano con `'_test.go'`. Successivamen-

te, eseguirà l'intera batteria di test, visualizzando l'output corrispondente direttamente nel terminale.

6 Fattori esterni e indipendenti dal framework

Al fine di sperimentare concretamente l'utilizzo del framework, è stato predisposto un server di prova, anch'esso incluso nella repository del progetto. Per la gestione delle dinamiche e delle route in questo server, si è fatto ricorso alla libreria **Gorilla Mux**, un pacchetto in Go che fornisce un router multiplexer (mux) per la gestione delle route nelle applicazioni web. Questa libreria semplifica notevolmente la gestione delle richieste HTTP e la mappatura degli URL a funzioni o handler specifici.

Il server, pur essendo totalmente dummy, consente la manipolazione di un database di utenti ed espone le seguenti route:

- `/user` con metodo "GET" per ottenere tutti gli utenti presenti nel database.
- `/user/id` con metodo "GET" per recuperare l'utente con l'id specificato come parametro.
- `/auth/user` con metodo "POST" per creare un nuovo utente, previa autenticazione, passando i parametri richiesti.

Oltre al server dedicato alla gestione diretta del database, è stato implementato un apigateway, che agisce come unico punto di ingresso per tutte le API. Pur offrendo numerose funzionalità tipiche degli API gateway, in questo contesto ne è stata fatta principalmente un'ampia adozione per due specifici motivi. La prima ragione è l'implementazione dell'autenticazione e dell'autorizzazione nelle API che richiedono tali funzionalità. A questo

scopo, è stato adottato il metodo di autenticazione mediante **JSON Web Token (JWT)**. Il JWT è uno standard aperto, definito dal RFC 7519, che stabilisce un formato compatto per la rappresentazione di informazioni tra due parti sotto forma di oggetto JSON. Questi token sono comunemente utilizzati per trasmettere informazioni in modo sicuro tra un client e un server, con la possibilità di essere firmati digitalmente per garantire l'integrità dei dati.

Un JWT è composto da tre parti separate da punti, ognuna codificata in Base64:

- **Header:** Contiene il tipo di token (typ) e l'algoritmo di firma (alg).
- **Payload:** Contiene le informazioni affermate dal token, comunemente chiamate "claims".
- **Firma:** Generata utilizzando l'header, il payload e una chiave segreta (o una coppia di chiavi pubblica/privata) con l'algoritmo specificato nell'header. La firma viene aggiunta al token e può essere utilizzata per verificare l'integrità del contenuto.

Il risultato è un token che può essere agevolmente scambiato tra le parti e verificato per garantire che non sia stato alterato. I JWT infatti, sono frequentemente impiegati per implementare l'autenticazione e l'autorizzazione in applicazioni web e servizi API.

All'interno della logica del gateway è stato implementato un rate limiter, facendo uso del pacchetto **golang.org/x/time/rate**, il quale fornisce un'implementazione di un algoritmo di rate limiting. Il rate limiting è una tecnica impiegata per controllare il numero di richieste o azioni che possono

essere eseguite in un determinato periodo di tempo. Questo particolare pacchetto è ampiamente utilizzato per attuare controlli di velocità o limitazioni di frequenza nelle applicazioni. Le componenti principali del pacchetto `golang.org/x/time/rate` includono:

- **Rate:** Il tipo `Rate` rappresenta un tasso di azioni consentite nel tempo. Può essere configurato con un numero massimo di azioni per secondo o per minuto, a seconda delle necessità.
- **Limiter:** Il tipo `Limiter` è un'istanza di un tasso che tiene traccia dello stato corrente e decide se un'azione è consentita in base al rate impostato. Un `Limiter` può essere utilizzato per controllare la frequenza con cui un'azione può essere eseguita.

Dal punto di vista progettuale, ogni route è dotata di un limiter impostato per un massimo di 10 richieste al secondo. Anche l'implementazione del gateway è presente nella repository del progetto.

Infine, per garantire una maggiore fluidità, l'intero sistema è orchestrato attraverso Docker Compose, il quale genera tre container dedicati ai tre servizi principali. Questa configurazione permette di gestire in maniera agevole l'ambiente di sviluppo, garantendo coerenza e facilitando la distribuzione dei servizi necessari all'interno dell'architettura del progetto.

What does API Gateway do?

 blog.bytebytego.com

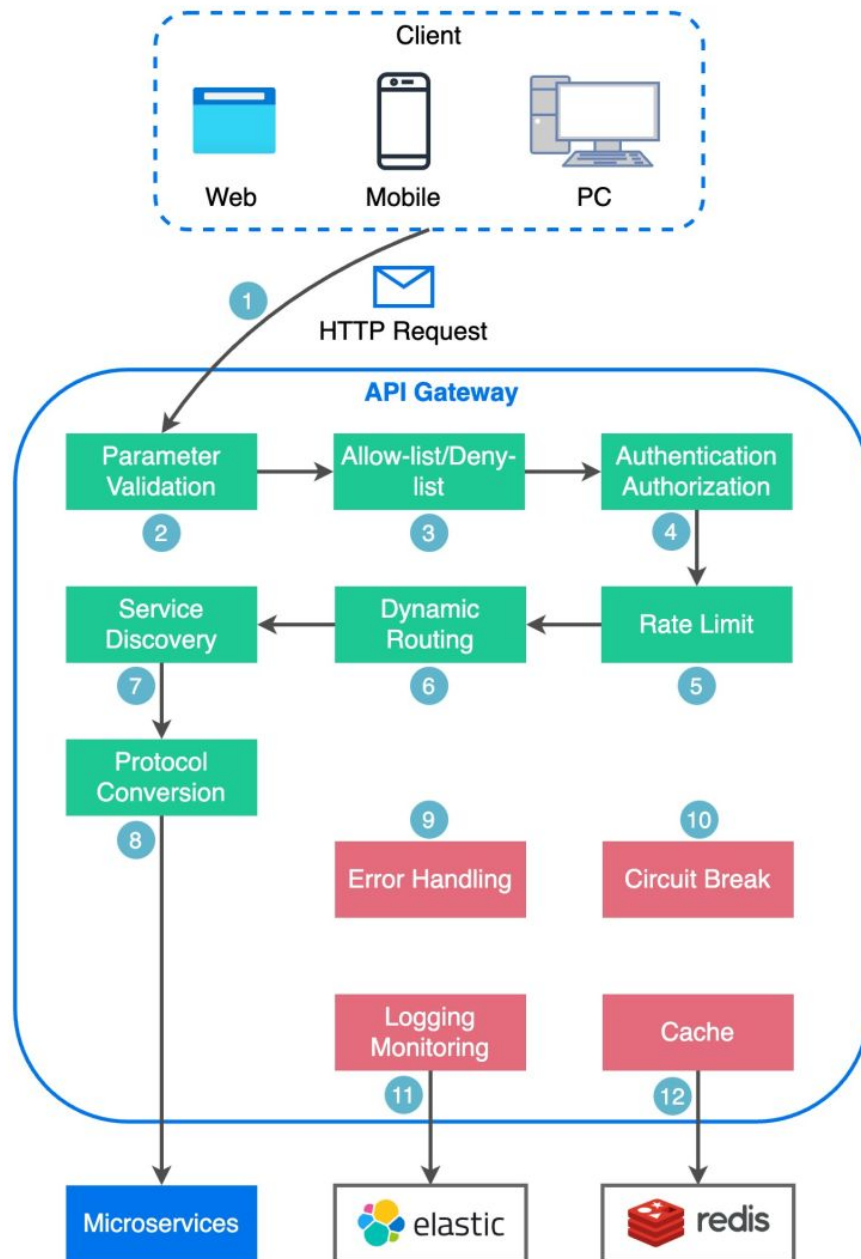


Figura 6: Panoramica delle possibili funzionalità offerte da un API gateway [4]

7 Architettura

di seguito è presentato un diagramma che illustra l'interazione tra l'utente e il framework per la creazione della test suite.

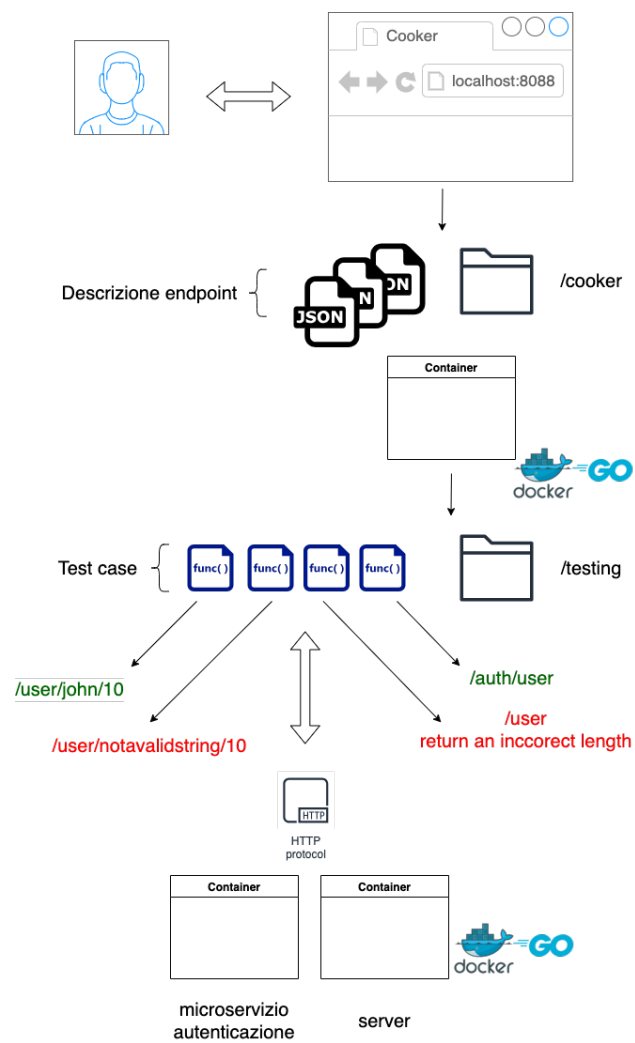


Figura 7: Architettura del progetto

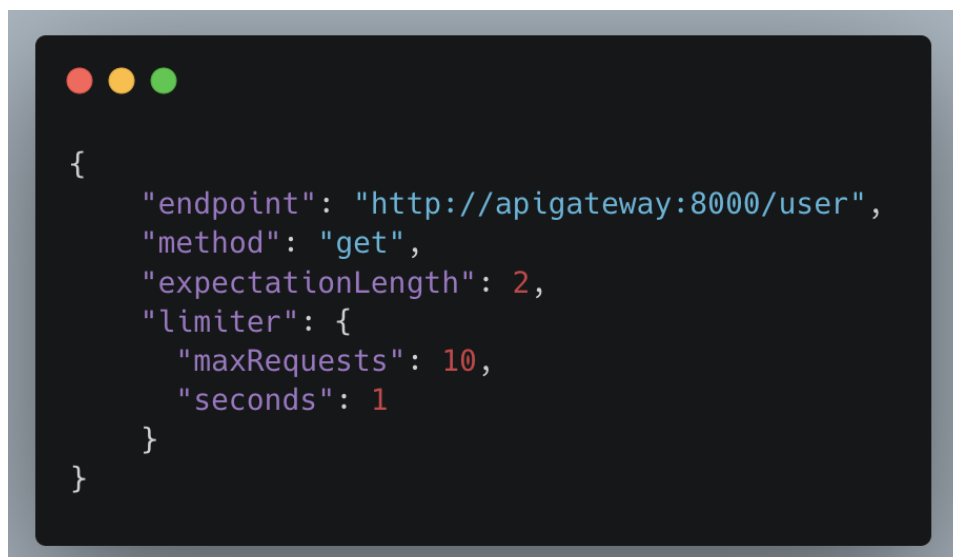
8 Esempi di utilizzo

8.1 Introduzione

In questa sezione, saranno illustrati tre esempi di utilizzo del framework. Saranno fornite le descrizioni degli endpoint sotto forma di JSON, corredate da una spiegazione dettagliata dei test generati per ciascuno di essi.

8.2 Esempio 1

Nella seguente descrizione dell'endpoint:



```
{
  "endpoint": "http://apigateway:8000/user",
  "method": "get",
  "expectationLength": 2,
  "limiter": {
    "maxRequests": 10,
    "seconds": 1
  }
}
```

Figura 8: Descrizione endpoint n°1

Oltre alle chiavi obbligatorie **"endpoint"** e **"method"**, viene specificato il valore della chiave **"expectationLength"**, che indica l'attesa di due valori di ritorno da questo endpoint. La presenza della chiave **"limiter"** indica la presenza di un limitatore lato server per questa route.

Questa sintetica descrizione permette la generazione di due test:

- Il primo test verifica se vengono restituiti effettivamente due valori e se lo status HTTP è 200 - OK.
- Il secondo test esegue 10 richieste entro un secondo e verifica se sono andate a buon fine. Successivamente, effettua la 11^a richiesta entro il secondo, la quale dovrebbe restituire il codice 429 - Too many requests. Infine, dopo aver superato il tempo necessario per il reset del limitatore, invia una nuova richiesta, aspettandosi nuovamente un esito positivo.

Numero totale di test generati: 2.

8.3 Esempio 2

Nella seguente descrizione dell'endpoint:



```
{
  "endpoint": "http://apigateway:8000/user",
  "method": "get",
  "parameters": [
    {
      "name": "id",
      "type": "uuid",
      "correctValue": "550e8400-e29b-41d4-a716-446655440000"
    }
  ],
  "expectationLength": 1,
  "limiter": {
    "maxRequests": 10,
    "seconds": 1
  }
}
```

Figura 9: Descrizione endpoint n°2

Anche in questo caso vengono fornite le chiavi obbligatorie "**endpoint**" e "**method**". Rispetto alla descrizione precedente, viene introdotto un parametro "**id**", che rappresenta l'identificativo di un record nel database. In questo caso, il numero di record attesi dalla route è uno. È presente anche la chiave "**limiter**", ma valgono le considerazioni fatte in precedenza.

Questa descrizione permette di generare i seguenti test:

- Il primo test verifica che, inviando l'UUID corretto, venga restituito un unico valore e che lo status HTTP sia 200 - OK.
- Il secondo test verifica che inviando un uuid non conforme, venga restituito un HTTP status 400 - Bad request.
- L'ultimo test, basato sulla verifica della correttezza del limitatore, segue le stesse considerazioni fatte per l'esempio precedente.

Numero totale di test generati: 3.

8.4 Esempio 3

Nella seguente descrizione dell'endpoint:

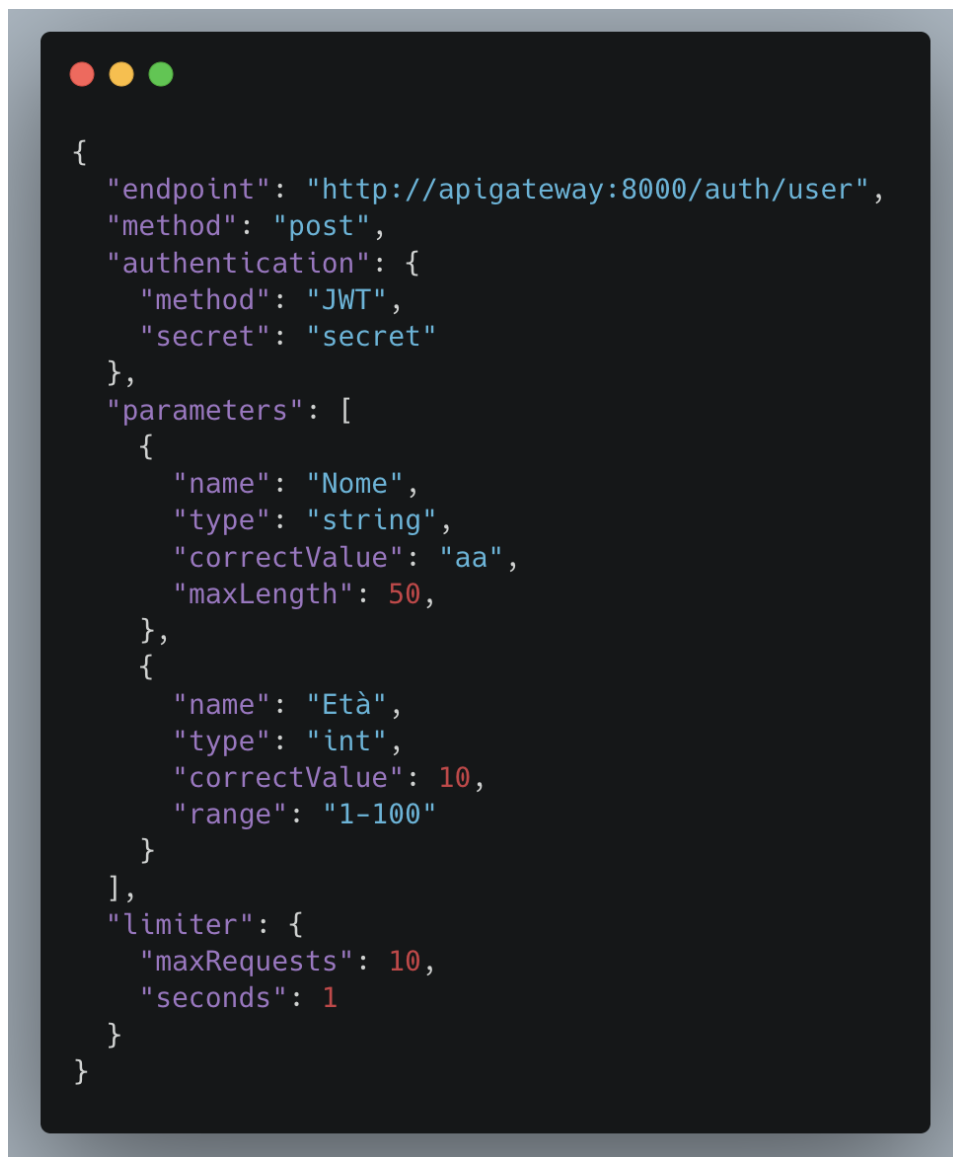


Figura 10: Descrizione endpoint n°3

In questa descrizione, tra le chiavi del JSON, troviamo la presenza della chiave **'authentication'**, il che significa che per ottenere i risultati desiderati è necessario effettuare un'autenticazione, fornendo un token bearer tra gli headers della richiesta. Per il momento, l'unico metodo accettato per effettuare autorizzazione e autenticazione è JWT, che richiede un **'secret'**,

anch'esso inviato come parametro. In tutti i test generati, verrà generato il token utilizzando il secret in questione, e per ogni richiesta sarà inserito tale token negli headers.

I test generati sono i seguenti:

- Il primo test verifica che, in assenza di autenticazione, la richiesta restituisca uno status HTTP 401 - Unauthorized.
- I successivi test sono di natura combinatoriale, ovvero provano tutte le possibili combinazioni di parametri. Solo la coppia di parametri corretti (forniti dal 'correctValue') darà come risposta uno status HTTP 201 - Created. In tutti gli altri casi, con coppie di parametri non corretti, ci si aspetterà come risposta uno status HTTP 400 - Bad Request.
- Infine, l'ultimo test, che verifica la correttezza del limitatore, segue le stesse considerazioni fatte per gli endpoint precedenti.

Numero totale di test: 9

9 Continuous Integration and Continuous Delivery (CI/CD)

Per tale framework è stato predisposto un workflow di Continuous Integration (CI) utilizzando GitHub Actions. Questo workflow permette di avviare tramite docker compose il framework Cooker e gli altri microservizi a contorno. Una volta che vengono generati i test, vengono eseguiti all'interno del workflow. Se tutti i test vengono verificati la pipeline sarà soddisfatta e la modifica proposta sarà integrata, nel caso in cui anche solo un test dovesse fallire, la pipeline sarà invalidata non procedendo con l'integrazione del codice. Vediamo nel dettaglio ogni step del workflow:

- **name:** Definisce il nome del workflow.
- **on:** Specifica gli eventi che attiveranno il workflow. In questo caso, il workflow viene eseguito quando ci sono push sul branch "main".
- **jobs:** Definisce i diversi jobs da eseguire nel workflow. In questo caso, c'è un unico job chiamato "test".
- **runs-on:** Specifica l'ambiente in cui eseguire il job. In questo caso, il job viene eseguito su una macchina virtuale con sistema operativo Ubuntu.
- **services:** Specifica i servizi da avviare per il job. In questo caso, viene avviato il servizio Docker con la versione 20.10.7.
- **steps:** Definisce i passaggi specifici da eseguire all'interno del job.
 - Checkout Repository: Clona il repository nella macchina virtuale.
 - Set up Docker Compose: Aggiorna i pacchetti e installa Docker Compose sulla macchina virtuale.
 - Build and run Docker Compose: Avvia i container Docker specificati nel file docker-compose.yml.
 - Execute Test: Esegue il comando per eseguire i test Go all'interno del container Docker "cooker".
 - Stop Docker Compose: Arresta i container Docker.

L'intera pipeline è possibile visualizzarla nella seguente immagine:



```
name: CI

on:
  push:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    services:
      docker:
        image: docker:20.10.7

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v3

      - name: Set up Docker Compose
        run: |
          sudo apt-get update
          sudo apt-get install -y docker-compose
          docker-compose --version

      - name: Build and run Docker Compose
        run: |
          docker-compose -f docker-compose.yml up -d

      - name: Excute Test
        run: |
          docker exec cooker go test ./...

      - name: Stop Docker Compose
        run: docker-compose -f docker-compose.yml down
```

Figura 11: Descrizione WorkFlow - Github Action

10 Futuri Sviluppi

Negli sviluppi futuri del framework, è imperativo ampliare in modo significativo la suite di test per ogni possibile endpoint. Questo implicherà l'integrazione di metodologie di testing che sono state solo accennate nel capitolo 4, ma che non sono state implementate nella versione iniziale del framework. Un obiettivo chiave è estendere le metodologie di autenticazione da sottoporre a test, nonché incorporare logiche di interrogazione per i metodi DELETE e PUT. Inoltre, si intende eseguire test approfonditi sull'idempotenza delle chiamate POST, contribuendo così a garantire la coerenza e la stabilità del sistema. Ci si propone anche di arricchire le chiavi di descrizione per ciascun tipo possibile, seguendo l'esempio positivo stabilito per i tipi string e int. Questo implicherà l'introduzione di parametri aggiuntivi come "maxLength" e "range" per fornire un controllo più dettagliato ai parametri inviati agli endpoint.

Inoltre in prospettiva delle prossime versioni, si prevede di implementare un ambiente di esecuzione dei test completamente isolato. Questa concezione si traduce nell'eliminare la necessità di importare i test nel progetto locale e di avviare manualmente la suite di test. L'approccio immaginato prevede che attraverso l'interfaccia, venga predisposto un modulo per l'esecuzione dei test in modalità cloud. Successivamente, sarà possibile generare un report dettagliato dell'esecuzione dei test, fornendo una panoramica chiara e informativa sullo stato e sulle prestazioni dei test effettuati.

Si sottolinea che il framework è open source, il che significa che è possibile contribuire. Le istruzioni dettagliate su come collaborare sono disponibili nel README della repository GitHub.

11 Conclusioni

La fase di testing si colloca da sempre come un pilastro fondamentale nello sviluppo di qualsiasi progetto. Tuttavia, spesso, questa fase viene trascurata a causa di vincoli temporali o risorse finanziarie limitate. In molti casi, gli sviluppatori si affidano a strumenti esterni nella speranza di velocizzare il processo di testing e ottenere una copertura del codice più ampia. In questo contesto, Cooker si presenta come un alleato affidabile per gli sviluppatori, offrendo un approccio gestionale intuitivo e accessibile. Il framework si propone di contribuire in modo significativo alla garanzia di robustezza e correttezza delle API lungo tutto il ciclo di sviluppo e manutenzione.

Riferimenti bibliografici

- [1] Skills And More - Rest e RestFul Api: <https://skillsandmore.org/rest-e-restful-api-introduzione/>
- [2] IT Consulting - Test Manuale vs Test Automatico: <https://www.itconsultingsrl.it/qual-e-la-differenza-tra-test-manuale-e-test-di-automazione/>
- [3] gRPC: <https://grpc.io/>
- [4] ByteByteGo - Design API - by Alex Xu: <https://blog.bytebytego.com/p/ep53-design-effective-and-safe-apis>
- [5] ByteByteGo - Testing API - by Alex Xu: <https://blog.bytebytego.com/p/ep83-explaining-9-types-of-api-testing>
- [6] Inetum - Why is software testing important <https://www.nearshore-it.eu/articles/why-is-software-testing-important/>

- [7] Medium - How Much Does Software Testing Cost? <https://hardiks.medium.com/how-much-does-software-testing-cost-9-proven-ways-to-optimize-it-a76275a04581>
- [8] JWT: <https://jwt.io/>
- [9] Ast - Golang: <https://pkg.go.dev/go/ast>
- [10] Rate Limiter - Golang: <https://pkg.go.dev/golang.org/x/time/rate>