

# Verifica TPS

## Identificare il tipo di applicazione

L'applicazione è una API REST che gestisce un database di utenti. Una API REST permette due applicazioni di comunicare tramite protocollo HTTP. Per essere RESTful deve rispettare diverse caratteristiche, tra cui:

- un webservice espone l'accesso a risorse di vario tipo (dati, procedure, dispositivi, sensori, ...) identificate mediante URL;
- le risorse sono rese disponibili mediante rappresentazioni che possono essere differenziate (per esempio: XML, JSON, ...);
- le operazioni che si effettuano sulle risorse sono quelle espresse dai metodi del protocollo HTTP (GET, POST, PUT, DELETE, ...) rispettandone la semantica;
- il webservice non memorizza lo stato dell'interazione con il client (stateless server) e, di conseguenza, ogni richiesta da parte del client deve contenere tutte le informazioni necessarie per essere servita;
- lo stato dell'interazione tra client e server è rappresentato e gestito mediante URL restituiti dal webservice al client, che identificano le rappresentazioni delle risorse: questo principio è definito HATEOAS (Hypermedia As The Engine Of Application State). Tutte queste proprietà sono rispettate dall'applicazione.

Questo software fa da middleware tra il client (es. sito web) e il server (contenente una istanza di un database come MongoDB che in questo caso è solo simulato da un dizionario di python). Utilizza quindi l'architettura client-server a 3 livelli, con il client che gestirà la User Interface, il middleware che gestirà la logica di business e il server che gestirà i dati.

Rispettando le regole delle API REST, le operazioni da eseguire sulle risorse sono specificate dal metodo HTTP e non nell'url, come di seguito dimostrato.

## Operazioni GET

### **Lista di tutti gli utenti del database**

The screenshot shows the Postman interface with a collection named "Verifica TPS". A new request is being built for the URL `http://127.0.0.1:5000/users`. The response body is displayed as a JSON array:

```
[{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]
```

## Richiesta di un utente specifico

The screenshot shows the Postman interface with a collection named "Verifica TPS". A new request is being built for the URL `http://127.0.0.1:5000/users/1`. The response body is displayed as a JSON object:

```
{"id": 1, "name": "Alice"}
```

## Operazioni POST

### Aggiunta di un nuovo utente

The screenshot shows the Postman interface with a dark theme. On the left, the sidebar displays 'My Workspace' with collections like 'AutoScraping' and 'Verifica TPS'. The main area shows a 'New Request' for a 'POST' method to 'http://127.0.0.1:5000/users'. The 'Body' tab is selected, containing the JSON payload: { "id": 3, "name": "Pietro"}. The response section shows a 201 CREATED status with a response body of { "id": 3, "name": "Pietro"}. Navigation icons at the bottom include 'Online', 'Find and replace', 'Console', 'Postbot', 'Runner', 'Start Proxy', 'Cookies', 'Vault', 'Trash', and a help icon.

This screenshot shows a 'GET' request to 'http://127.0.0.1:5000/users/3' under the 'Verifica TPS' collection. The 'Headers' tab is selected, showing 'Content-Type: application/json'. The 'Query Params' table has one entry: Key 'id' with Value '3'. The response section shows a 200 OK status with a response body of { "id": 3, "name": "Pietro"}. The same navigation icons as the first screenshot are visible at the bottom.

# Operazioni PUT

## Aggiornamento di un utente

The screenshot shows the Postman application interface. On the left, the sidebar displays 'My Workspace' with collections like 'AutoScraper' and 'Verifica TPS'. The main area shows a 'PUT Update User' request. The request URL is `http://127.0.0.1:5000/users/1`. The 'Body' tab is selected, showing raw JSON data: 

```
1 { "id": 1, "name": "Francesca"}
```

. The response status is 200 OK with a 3 ms duration and 202 B size. The response body is identical to the sent body.

The screenshot shows the Postman application interface. On the left, the sidebar displays 'My Workspace' with collections like 'AutoScraper' and 'Verifica TPS'. The main area shows a 'GET Specific User' request. The request URL is `http://127.0.0.1:5000/users/1`. The 'Body' tab is selected, showing raw JSON data: 

```
1 { "id": 1, "name": "Francesca"}
```

. The response status is 200 OK with a 5 ms duration and 202 B size. The response body is identical to the sent body.

## Operazioni DELETE

### Eliminazione di un utente

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'My Workspace' containing collections like 'AutoScraper' and 'Verifica TPS'. The main area displays a DELETE request to 'http://127.0.0.1:5000/users/1'. The 'Body' tab is selected, showing the option 'none' is chosen. Below the request, the response status is '200 OK' with a response time of 7 ms and a body size of 189 B. The response content is 'Utente eliminato'.

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'My Workspace' containing collections like 'AutoScraper' and 'Verifica TPS'. The main area displays a GET request to 'http://127.0.0.1:5000/users'. The 'Headers' tab is selected, showing 'Content-Type: application/json'. The 'Body' tab is selected, showing a JSON response with one item: [ { "id": 2, "name": "Bob" } ].

# Docker

È possibile far girare l'applicazione in un docker container per una migliore portabilità e scalabilità.

Di seguito un esempio di DockerFile. (utilizzerò poetry per la gestione del virtual environment)

```
# Use an official Python image
FROM python:3.13-slim

# Install Poetry
RUN python -m pip install poetry

# Set working directory
WORKDIR /app

# Copy Poetry files
COPY poetry.lock pyproject.toml ./

# Install dependencies
RUN poetry install --no-interaction --no-ansi --no-root

# Copy the application
COPY . .

# Install the application
RUN poetry install --no-interaction --no-ansi

# Expose the port Flask runs on
EXPOSE 8000

# Start the FastAPI server
CMD ["poetry", "run", "python", "applicazione.py"]
```

Per creare l'immagine:

```
sudo docker build .
```

Per runnare il container:

```
sudo docker run -d -p 5000:5000 <container_id>
```

## Docker compose

```
version: "3.8"

services:
  app:
    build: .
    ports:
      - "5000:5000"
      - ./app
    restart: unless-stopped
```

Per runnarlo:

```
sudo docker compose up --build -d
```

## Ulteriori possibilità

Volendo automatizzare la scalabilità dell'applicazione potrei creare un cluster di docker container con docker swarm.

## Framework

Il framework utilizzato per l'implementazione delle API è `flask`.

**Flask** è un **micro web framework** scritto in Python, utilizzato per sviluppare applicazioni web in modo semplice e veloce.

[Fonte][[https://en.wikipedia.org/wiki/Flask\\_\(web\\_framework\)](https://en.wikipedia.org/wiki/Flask_(web_framework))]

## Caratteristiche principali:

- **Leggero e minimalista:** fornisce solo gli strumenti essenziali, lasciandoti la libertà di scegliere librerie e componenti aggiuntivi.
- **Routing facile:** permette di associare facilmente URL a funzioni Python.
- **Supporta JSON, form HTML, richieste HTTP...**
- **Modulare:** puoi espanderlo con estensioni (es. per database, login, ecc.).
- **Perfetto per piccoli progetti e REST API.**

## Utilizzi

Flask viene utilizzato negli ambiti della sanità, dei viaggi e della finanza.

## Viaggi

Flask è l'ideale per applicazioni basate sull'intelligenza artificiale, come motori di ricerca, route planners, e sistemi di pricing dinamici.

## Sanità

Flask è utilizzato in applicazioni che usano l'IA per le diagnosi mediche, chatbot medici e piattaforme di sanità telematica. Un solido esempio di questa applicazione è [FlaskData.io] [\_FlaskData.io\_].

## Finanza

Flask è ideale per applicazione di tracking delle spese e applicazioni di rilevamento delle truffe che usano l'IA.

[Fonte][<https://embarkingonvoyage.com/blog/technologies/how-python-flask-django-for-enterprise-applications-are-powering-digital-innovation/>]

## Alternative

Una valida alternativa a flask è il framework `FastAPI`.