



REST & Python



Costruire una api rest in
Python

Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia



Costruire una api rest in Python e Flask

Architettura REST

Architettura REST

{ **REST:API** }

- **R**epresentational **S**tate **T**ransfer
- Permettere azioni **CRUD** su sorgente dati remota
- mappa azioni **CRUD** su metodi **http**

metodo http	operazione
POST	create
GET	read
PUT	update
DELETE	delete

Architettura REST

response standard

{ **REST:API** }

- l'architettura REST prevede che la **response** abbia una struttura standard come quella sotto indicata in un esempio

```
{  
  "success": true,  
  "message": "User logged in successfully",  
  "data": { }  
}
```

REST RESPONSE

- nel caso di successo dell'operazione il valore della chiave **success** varrà **true**, **false** altrimenti
- la chiave **message** contiene un messaggio *human-friendly* che descrive cosa sia successo
- la chiave **data** contiene i dati ricevuti (un oggetto JSON che contiene le informazioni richieste)

Architettura REST - - esempi

- richiesta **[URI]/tasks** per recuperare una lista di oggetti di tipo *task*

{ **REST:API** }

REST RESPONSE

```
{
  "success": true,
  "message": "tasks presenti nell'archivio",
  "data": {
    "tasks": [
      {
        "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
        "done": false,
        "id": 1,
        "title": "Buy groceries"
      },
      {
        "description": "Need to find a good Python tutorial on the web",
        "done": false,
        "id": 2,
        "title": "Learn Python"
      }
    ]
  }
}
```

Architettura REST - esempi

- richiesta **[URI]/oggetti non presenti** che punta ad un *endpoint* sbagliato

{ **REST:API** }

REST RESPONSE

```
{
  "success": false
  "message": "Servizio non presente",
  "error_code": 1000,
  "data": [],
}
```

- richiesta **[URI]/tasks/100** che punta ad un *endpoint* di inserimento task con id=100 (metodo POST)

REST RESPONSE

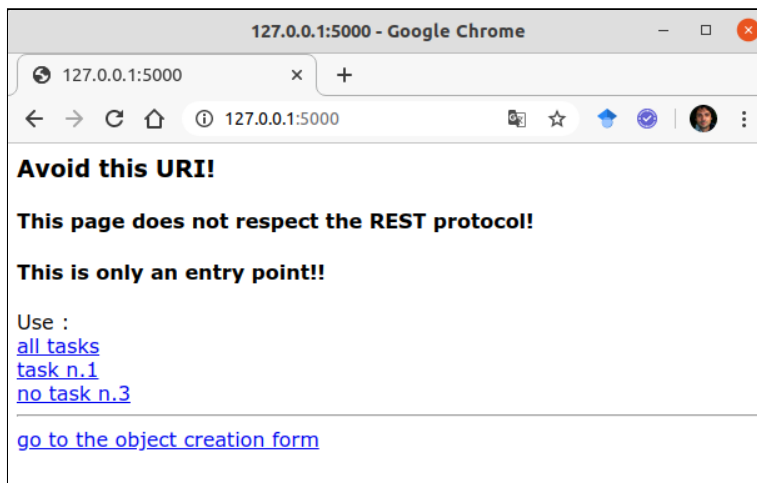
```
{
  "success": true
  "message": "Task inserito correttamente",
  "data": {
    "task" : {
      "description": "Usare flask e python per creare servizi REST",
      "done": false,
      "id": 100,
      "title": "Web Services REST"
    }
  },
}
```

Architettura REST

Architettura REST

{ **REST:API** }

- creiamo con *Flask* un servizio esposto all'URI *http://127.0.0.1:5000* per esporre dati di *tasks* da eseguire
- **N.B.** : la chiamata all'URI *http://127.0.0.1:5000* restituisce l'home page che **NON** rispetta lo standard rest ma è inserita solo per semplicità d'uso, infatti restituisce HTML e non JSON come prevede lo standard



Leggere dati di risorse

- operazione di lettura **READ** è mappata su metodo **GET** per sapere tutti i task presenti
 - URI: `http://127.0.0.1:5000/tasks`
 - metodo: **GET**; body: *nessuno*

RESPONSE

```
{
  "data": {
    "tasks": [
      {
        "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
        "done": false,
        "id": 1,
        "title": "Buy groceries"
      },
      {
        "description": "Need to find a good Python tutorial on the web",
        "done": false,
        "id": 2,
        "title": "Learn Python"
      }
    ]
  },
  "message": "Lista dei task presenti",
  "success": true
}
```

- STATUS: **200** // tutto OK

Dietro le quinte - lato server

- micro server in python e flask framework

PYTHON

```
from flask import Flask, jsonify, abort, make_response, request
app = Flask(__name__)
app.secret_key = "12345678901234567890"

# simuliamo il database con una lista di oggetti
tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the web',
        'done': False
    }
]

<qui inseriamo le funzioni che mappano le chiamate>

if __name__ == '__main__':
    app.run()
```

Dietro le quinte - lato server

- ogni funzione che scriviamo mappa una chiamata grazie al *decoratore* **@app.route** (1)
- la chiamata **/tasks** viene mappata sulla funzione **tasklist()**

PYTHON

```
# READ => GET
@app.route('/tasks', methods=['GET'])
def tasklist():

    response_dict = {
        "success": True,
        "message": "Lista dei task presenti",
        "data": {
            "tasks": tasks
        }
    }

    body_response = json.dumps(response_dict) # trasformo il dizionario in oggetto JSON
    response = make_response(body_response)   # creo l'oggetto "response" per inviare la risposta
    # setto il content-type corretto (altrimenti di default è text/html)
    response.headers["Content-type"] = "application/json"
    return response # rispondo al client
```

Dietro le quinte - lato server (2)

- flask mette a disposizione la funzione **jsonify** per questo scopo
- jsonify setta automaticamente anche il content-type della *response* a **application/json**

```
@app.route('/tasks', methods=['GET'])
def taskslis():
    response_dict = {
        "success": True,
        "message": "Lista dei task presenti",
        "data": {
            "tasks": tasks
        }
    }
    return jsonify(response_dict), 200
```

PYTHON

Dietro le quinte - lato server (3)

PYTHON

```
@app.route('/tasks', methods=['GET'])
def taskslis():
    [...]
    return jsonify(response_dict), 200
```

- nel decoratore è possibile specificare i *metodi* http accettati
- in questo caso solo il metodo **GET** perché dobbiamo performare una operazione **READ**
- il metodo *jsonify* trasforma una lista di python in una lista in formato JSON (in questo caso il formato in realtà coincide...)
- la funzione *taskslis* restituisce 2 valori:
 - il body della risposta : la lista in formato JSON
 - lo status code dell'operazione : 200 // che corrisponde a **HTTP/1.0 200 OK**

1) Un decoratore è una funzione che *modifica* a runtime un'altra funzione prima che questa venga eseguita, aggiungendo o modificando i suoi comportamenti. Rientra nella logica della **metaprogrammazione**

Creazione di un nuovo task

- dobbiamo sapere l'**endpoint** per l'operazione *insert*
- dobbiamo usare il metodo POST
- dobbiamo inviare nel body i dati necessari all'operazione
- nel caso specifico:
 - URI: **http://127.0.0.1:5000/task**

PYTHON

```
@app.route('/task', methods=['POST']) # risponde solo alla chiamata con metodo POST
def endpoint_per_creazione_task_via_post():
    """
        creo un nuovo TASK - qui dovrei accedere al database
        prendo l'id successivo all'ultimo task creato
        uso i dati della request.form (ovvero POST) per valorizzare il task
    """
    task = {
        'id': tasks[-1]['id'] + 1,
        'title': request.form['title'],
        'description': request.form.get('description', ""),
        'done': False
    }
    tasks.append(task)
    response_success_template["message"] = "Task creato correttamente"
    response_success_template["data"] = {"task": task}
    return jsonify(response_success_template), 201 # 201 è lo stato OK CREATE di http
```

Creazione di un nuovo task

- la chiamata http deve effettuare un POST all'*endpoint*
- deve includere nel body delle request i campi per passare i valori necessari

The screenshot displays a REST client interface with the following components:

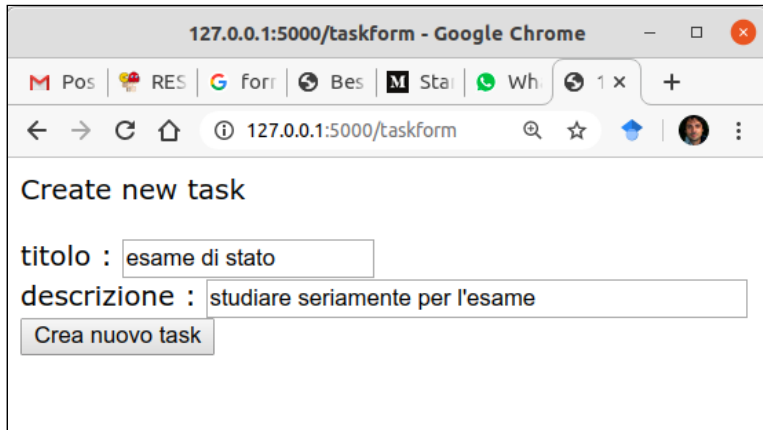
- Request Bar:** Method **POST**, URL **http://127.0.0.1:5000/task**, and a **Send** button.
- Response Status Bar:** **201 CREATED** (in green), **8.34 ms**, **162 B**, and **5 Minutes Ago**.
- Form Tab (Active):** Contains a table for request body fields:

Field	Value	Actions
title	usare rest	Dropdown, Check, Delete
description	imparare REST è necessario	Dropdown, Check, Delete
New name	New value	
- Preview Tab (Active):** Displays the JSON response body:

```
1 {
2   "data": {
3     "task": {
4       "description": "imparare REST è necessario",
5       "done": false,
6       "id": 3,
7       "title": "usare rest"
8     }
9   },
10  "message": "Task creato correttamente",
11  "success": true
12 }
```

Creazione di un nuovo task

- la stessa chiamata potrei farla tramite form in una pagina HTML



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:5000/taskForm - Google Chrome'. The page content includes the heading 'Create new task', followed by two input fields: 'titolo : esame di stato' and 'descrizione : studiare seriamente per l'esame'. Below these fields is a button labeled 'Crea nuovo task'.

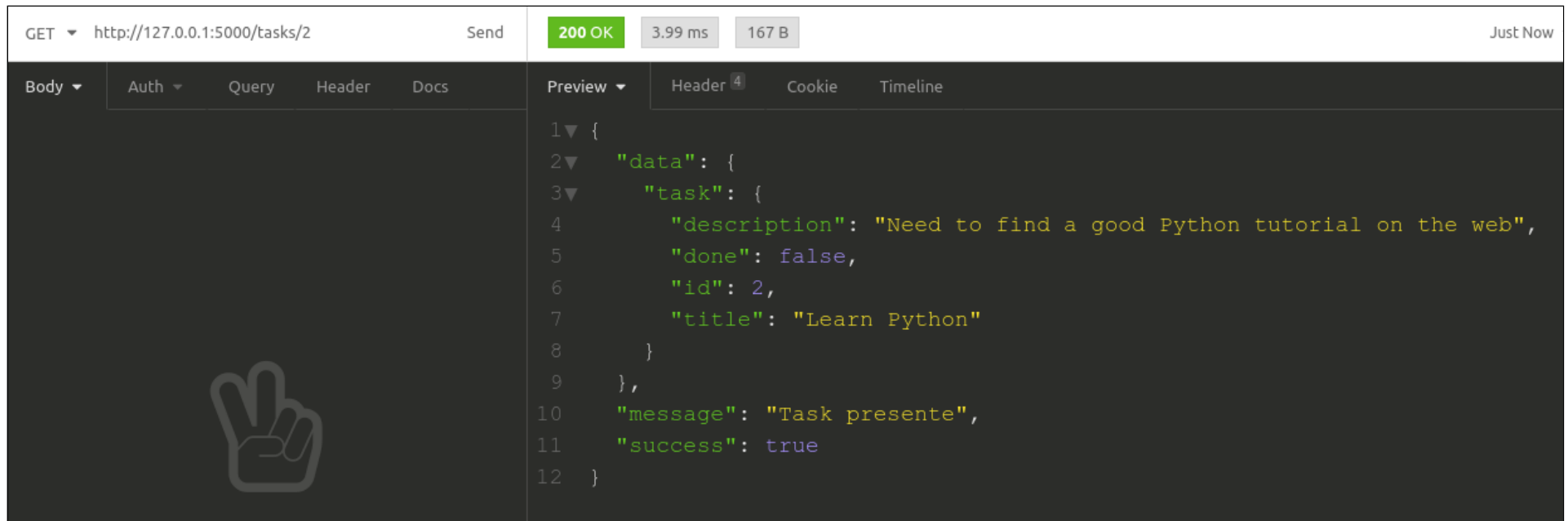
il cui codice HTML è

```
<form action='/task' method='POST'>
  titolo : <input name='title' value='esame di stato'><br>
  descrizione : <input size=50 name='description' value="studiare seriamente per l'esame"><br>
  <input type='submit' name='submit' value='Crea nuovo task'>
</form>
```

HTML

Ricerca di un singolo task per id

- dobbiamo sapere l'**endpoint** per l'operazione *read*
- dobbiamo usare il metodo GET - nessun body
- nel caso specifico:
 - URI: **http://127.0.0.1:5000/tasks/<task id>**
 - es: http://127.0.0.1:5000/tasks/2



GET http://127.0.0.1:5000/tasks/2 Send 200 OK 3.99 ms 167 B Just Now

Body Auth Query Header Docs Preview Header 4 Cookie Timeline

```
1 {
2   "data": {
3     "task": {
4       "description": "Need to find a good Python tutorial on the web",
5       "done": false,
6       "id": 2,
7       "title": "Learn Python"
8     }
9   },
10  "message": "Task presente",
11  "success": true
12 }
```


Lato server...

- La funzione che intercetta la chiamata **[URI]/tasks/<task id>** è

PYTHON

```
@app.route('/tasks/<int:task_id>', methods=['GET']) # solo GET
def task(task_id):
    for elem in tasks: # scorro la lista => select con where su DB
        if elem['id'] == task_id:
            response_success_template["message"] = "Task presente"
            response_success_template["data"] = {"task" : elem}
            return jsonify(response_success_template), 200
    # default => not found
    response_error_template["message"] = "Task non presente"
    response_error_template["error_code"] = 5 # un codice applicativo qualsiasi documentato
    return jsonify(response_error_template), 404
```

Web service REST con python e flask

- esercizi per lo studente
- implementare le funzioni DELETE e UPDATE
- spostare i dati da dizionario a DB
- il codice di partenza visto in questa lezione è disponibile su



Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia