How to build a Book-API with Python and Flask

Part 1 - Theoretical Part

What is a REST API?

REST is an architectural style for designing networked applications. RESTful APIs use HTTP methods (e.g. GET, POST, PUT, DELETE) to perform various operations on resources. Here's a brief explanation of each HTTP method in the context of a REST API:

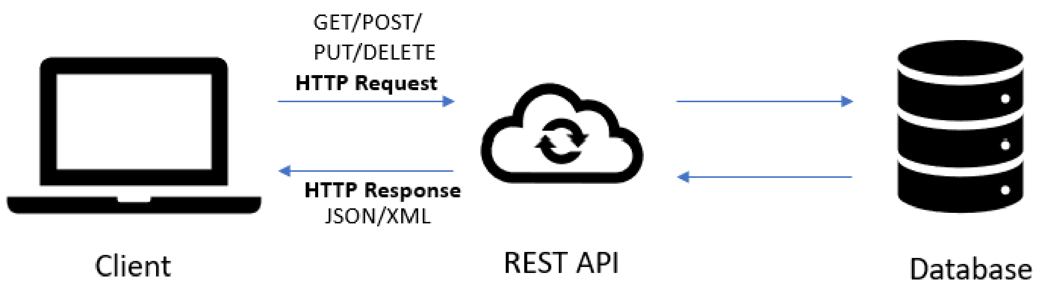
- GET: Used to retrieve data from the server. It doesn't modify data on the server.
- POST: Used to create new resources on the server. It submits data to be processed to a specified resource.
- PUT: Used to update existing resources on the server.
- DELETE: Used to delete resources from the server.

What are Response codes (HTTP Status Codes)?

HTTP status codes are three-digit numbers returned by a web server to indicate the outcome of a client's request. They convey information about whether a request was successful, encountered an error, or needs further action. Some common HTTP status codes include:

- 200 (OK): The request was successful.
- 201 (Created): A new resource was successfully created.
- 404 (Not Found): The requested resource was not found.
- 400 (Bad Request): The request was malformed or had invalid data.
- 500 (Internal Server Error): An unexpected server error occurred.

How can a simplified workflow look like?



- 1. Client Sends Request: The client, which can be a web browser, mobile app, or any application, sends an HTTP request to a REST API. The HTTP request methods commonly used are GET, POST, PUT, or DELETE.

- 2. REST API Receives Request: The REST API built using a framework like Flask (in Python) receives the HTTP request. The API processes the request and performs operations based on the HTTP method used.
- 3. API Interacts with Database: In many real-world scenarios, a REST API interacts with a database. This interaction includes operations like querying for data (GET), inserting new data (POST), updating existing data (PUT), or deleting data (DELETE). The API translates the HTTP request into appropriate database operations.
- 4. Database Operations: The database, which can be SQL-based (e.g., MySQL, PostgreSQL) or NoSQL-based (e.g., MongoDB, Redis), processes the requests from the API. It performs the required database operations like retrieving, creating, updating, or deleting records.
- 5. Database Sends Response to API: After performing the operations, the database sends back a response to the REST API. This response typically includes the data retrieved or a confirmation of the successful execution of the requested operation.
- 6. REST API Formats Response: The REST API formats the response data, often into JSON or XML, and includes appropriate HTTP status codes. The status codes indicate whether the operation was successful (e.g., 200 OK) or encountered an error (e.g., 404 Not Found).
- 7. API Sends Response to Client: The REST API sends the formatted response, including the data and status codes, back to the client that made the initial request.
- 8. Client Receives Response: The client application receives the response from the API. It can then process the data received or take appropriate actions based on the response and status codes.

How can we build APIs with Python and Flask?

Flask is a micro web framework for building web applications and APIs in Python. Flask makes it easy to define routes and associate them with specific functions (called view functions) that execute when a particular URL is accessed. Developers can use Python functions to define how requests should be handled. This simplifies the mapping of HTTP methods (GET, POST, PUT, DELETE) to specific actions.

Part 2 - Technical Part

Requirement:

- Build a Book API which allows you to perform CRUD (Create, Read, Update, Delete) operations
- Use Python and Flask (lightweight web framework for Python that simplifies the creation of web apps, including building APIs)

Import the necessary packages

```
In [4]: from flask import Flask,request,jsonify
```

Flask web application instance

```
In [ ]: app = Flask(__name__)
```

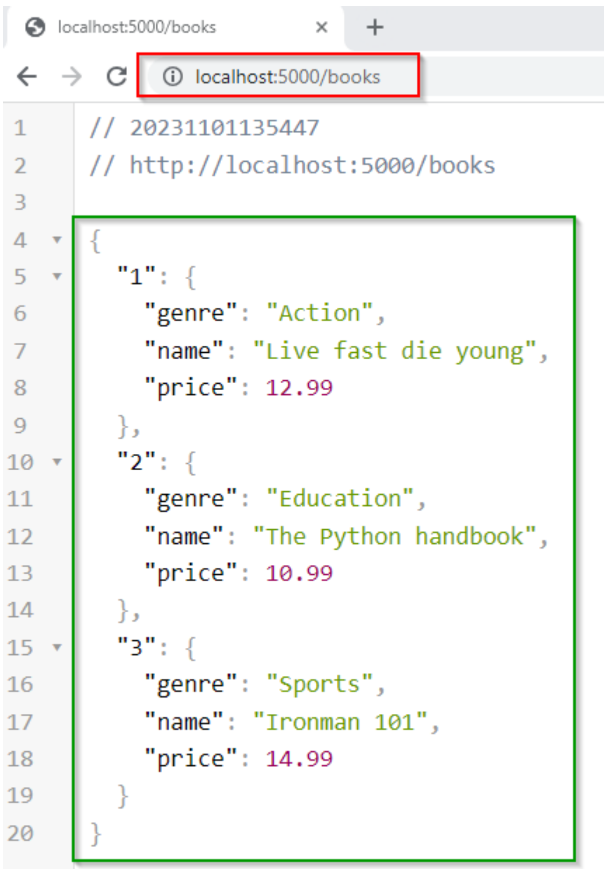
Create database/record of Books (dictionary format)

- Note: in real projects it is common to integrate a database. For simplicity reasons, a dictionary will be used as our database

```
In [ ]: books = {
            1: {"name": "Live fast die young", "genre": "Action", "price": 12.99},
            2: {"name": "The Python handbook", "genre": "Education", "price": 10.99},
            3: {"name": "Ironman 101", "genre": "Sports", "price": 14.99}
        }
```

GET-Endpoint (Fetch all Books)

```
In [ ]: # Decorator with "/books" endpoint - responds only to get-requests
        # function is executed when a GET request is made
        # HTTP status code is 200 (=OK, request was successful)
        @app.route('/books', methods=['GET'])
        def get_books():
            return jsonify(books), 200
```

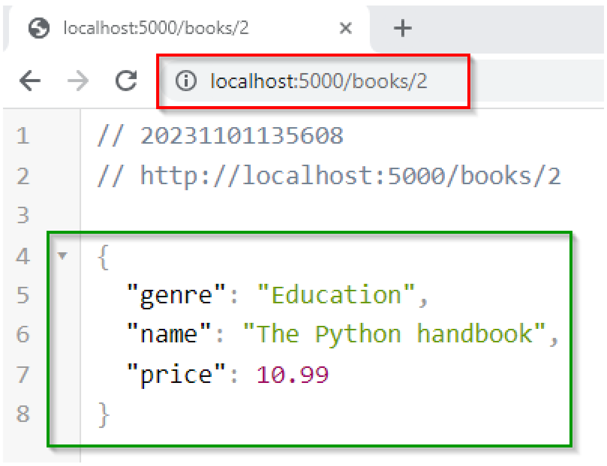


```
// 20231101135447
// http://localhost:5000/books

{
  "1": {
    "genre": "Action",
    "name": "Live fast die young",
    "price": 12.99
  },
  "2": {
    "genre": "Education",
    "name": "The Python handbook",
    "price": 10.99
  },
  "3": {
    "genre": "Sports",
    "name": "Ironman 101",
    "price": 14.99
  }
}
```

GET-Endpoint (Fetch Books by ID)

In [ ]:
```python
# Endpoint includes a dynamic parameter
# function which accepts a book id as a parameter
@app.route('/books/<int:book_id>', methods=['GET'])
def get_book(book_id):
    # attempt to retrieve a book from books-dictionary
    book = books.get(book_id)
    # if the specific book id is present, user gets the record back,
    # HTTP Code 200 (OK)
    if book:
        return jsonify(book), 200
    # if the book id is not present, user gets back the
    # HTTP Code 404 (Not Found)
    else:
        return jsonify({"message": "Book not found"}), 404
```



```
// 20231101135608
// http://localhost:5000/books/2

{
  "genre": "Education",
  "name": "The Python handbook",
  "price": 10.99
}
```

POST-Endpoint (Add a new Book)

In [ ]:
```python
# Decorator for POST requests and endpoint "/books"
@app.route('/books', methods=['POST'])
# function is executed when POST request is made to this endpoint
def add_book():
    data = request.json
    # retrieve book id
    book_id = data.get("id")
    # retrieve book name
    name = data.get("name")
    # retrieve book genre
    genre = data.get("genre")
    # retrieve book price
    price = data.get("price")

    # if book already exists, nothing will be added to the dictionary
    # HTTP Code 400 (Bad Request)
    # if book does not exist,a new entry will be made - HTTP Code 201 (Created)
    if book_id and name and genre and price is not None:
        if book_id in books:
            return jsonify({"message": "Book ID already exists"}), 400
        else:
            new_book = {"name": name, "genre": genre, "price": price}
            books[book_id] = new_book
            return jsonify({"message": f"Book '{name}' added"}), 201
    # if data attributes are incomplete or wrong data is provided
    # HTTP Code 400 (Bad Request)
    else:
        return jsonify({"message": "Invalid data"}), 400
```

PUT-Endpoint (Update a Book record)

In [ ]:
```
# Decorator for PUT requests and dynamic endpoint "/books/<int:book_id>"
@app.route('/books/<int:book_id>', methods=['PUT'])
# fucntion takes a book id as a parameter
def update_book(book_id):
    data = request.json
    # retrieve name
    name = data.get("name")
    # retrieve genre
    genre = data.get("genre")
    # retrieve price
    price = data.get("price")

    # checks if book id exists in the record
    book = books.get(book_id)
    # if book exists, name, genre and price will be updated
    if book:
        if name is not None:
            book["name"] = name
        if genre is not None:
            book["genre"] = genre
        if price is not None:
            book["price"] = price
        # after updating the book, HTTP code 200 is returned (OK)
        return jsonify({"message": f"Book with ID {book_id} updated"}), 200
    # if book doe not exists, we get back an HTTP 404 Code (Not Found)
    else:
        return jsonify({"message": "Book not found"}), 404
```

DELETE-Endpoint (Delete a Book record)

In [ ]:
```
# Delete a book by ID
# Decorator for Delete requests and dynamic endpoint "/books/<int:book_id>"
@app.route('/books/<int:book_id>', methods=['DELETE'])
# function is executed when Delete request is made (book id as a parameter)
def delete_book(book_id):
    # if the book id exists, delete the book entry
    if book_id in books:
        del books[book_id]
        # HTTP Code 200 (OK)
        return jsonify({"message": f"Book with ID {book_id} deleted"}), 200
    # if book id does not exists, HTTP Code 404 (Not Found) will be given back
    else:
        return jsonify({"message": "Book not found"}), 404
```

Start Flask application and set up localhost and port

In [ ]:
```
# starts the Flask application and specifies that it should run on the
# local host (your computer) with port 5000
app.run(host='localhost', port=5000)
```