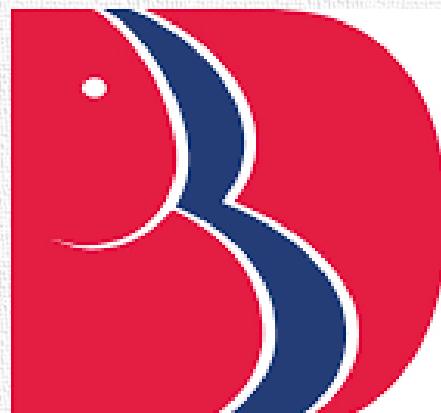


# SCHOOL OF COMPUTER APPLICATION

CLOUD APPLICATION DEVELOPMENT

BCADSN12101



**BBD UNIVERSITY**

Submitted By:

**Name: Shantanu Pandey**

**Section: BCA DS 16**

**Roll No: 05**

**Semester: 2<sup>nd</sup>**

TO

**Mr. Nikhil Sharma**

**Babu Banarasi Das University, Lucknow**

**BBD City, Ayodhya Road, Lucknow Uttar Pradesh- 226028 Bharat**

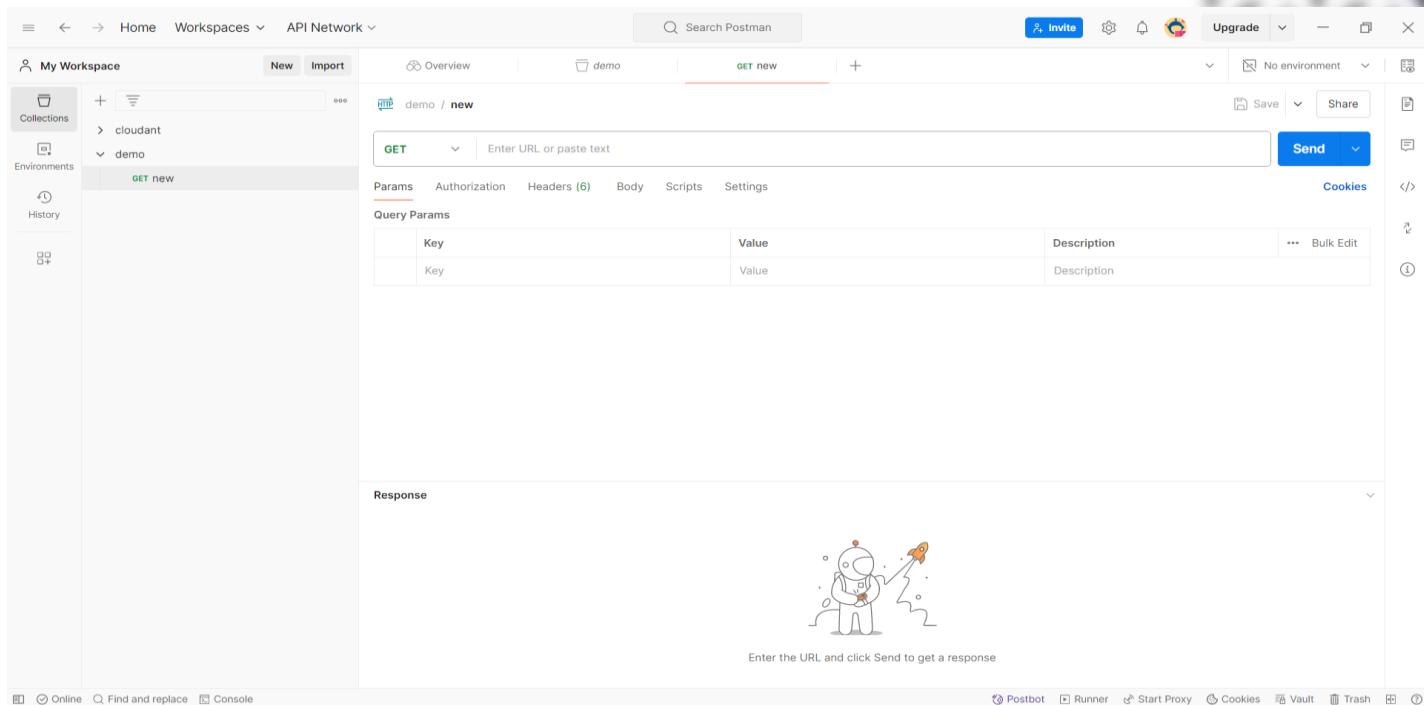
# INDEX

S.NO	TITLE	DATE	PAGE NO
1	TEXT TO SPEECH	MARCH 10 2024	1 TO 4
2	SPEECH TO TEXT	MARCH 10 2024	5 TO 7
3	LANGUAGE TRANSLATOR	MARCH 26 2024	8 TO 27
4	DOCKER IMAGE BUILDING	APRIL 28 2024	28 TO 39
5	REST API IMPLEMENTATION	MAY 11 2024	40 TO 51

# REST API IMPLEMENTATION USING FLASK

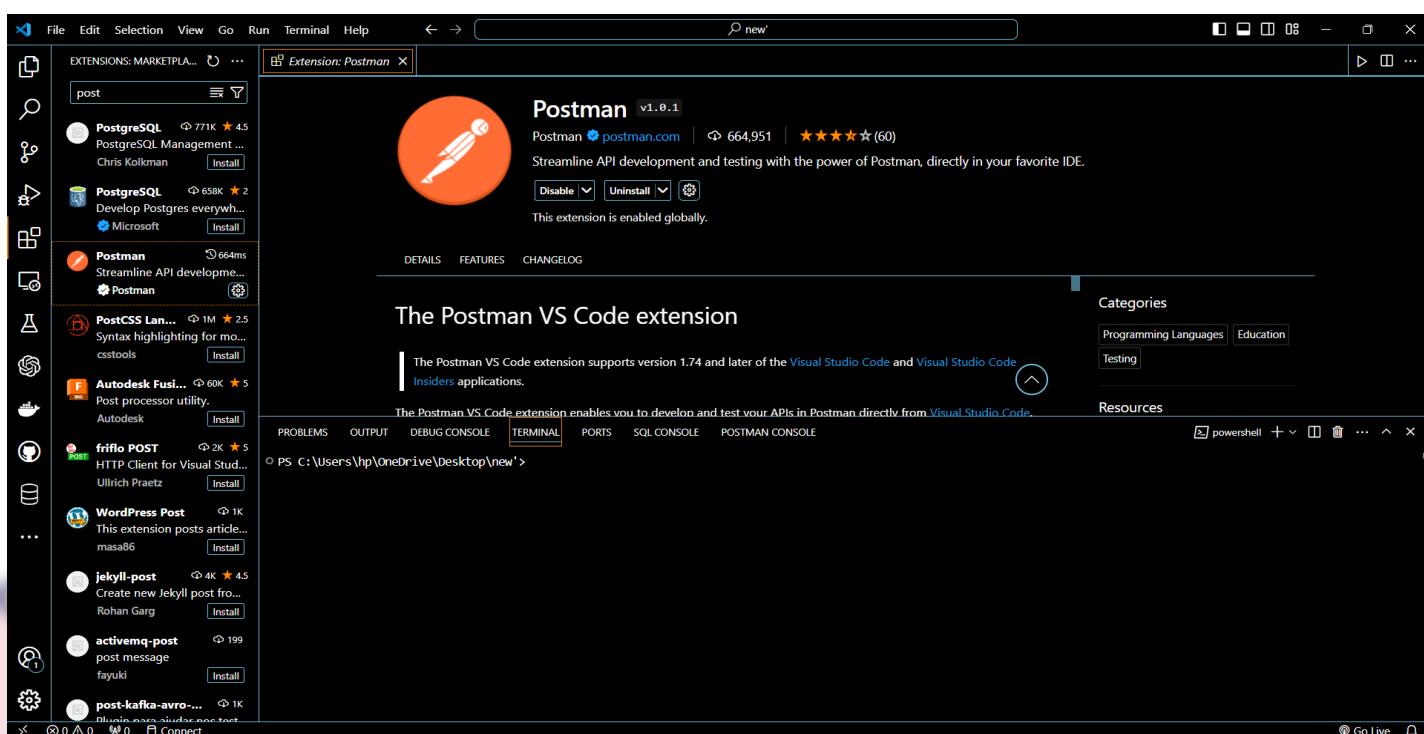
## STEP 1 - Install POSTMAN application

(<https://www.postman.com/downloads/>) and ensure that you have installed Postman correctly and running on your system. Sign-in with your credentials.

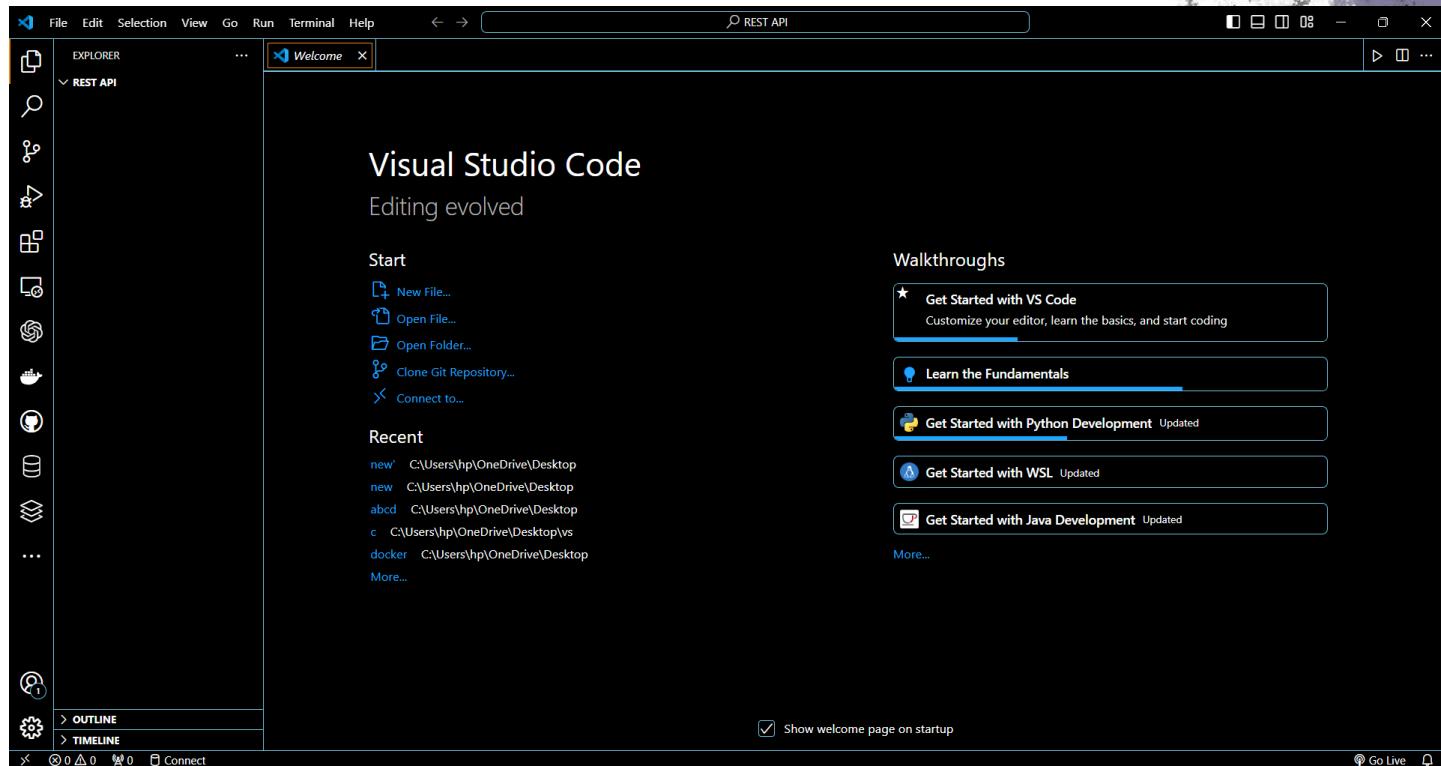
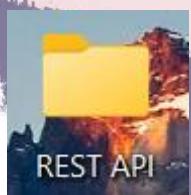


## STEP 2 - Install VS Code and the official Postman extension from the VS Code Marketplace.

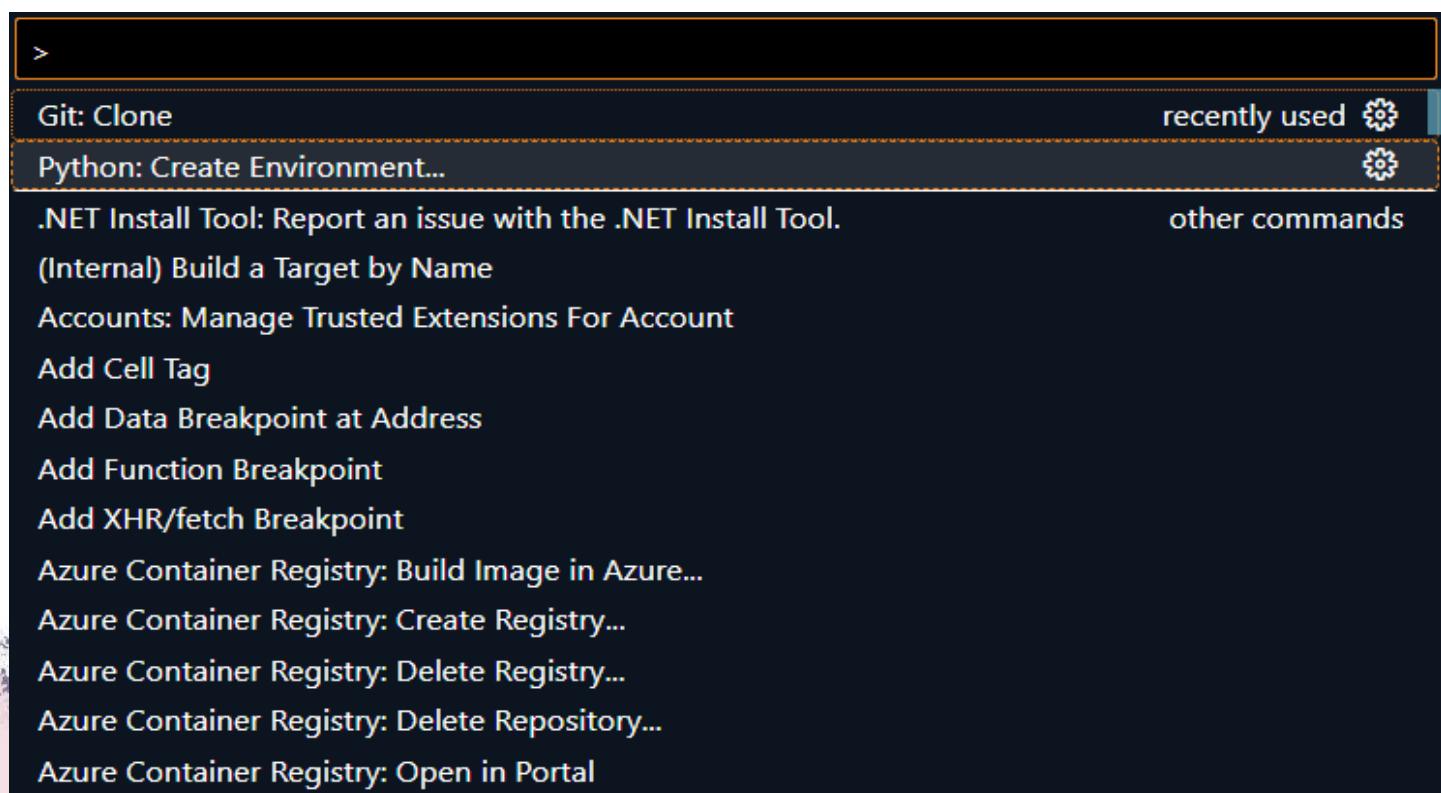
(<https://code.visualstudio.com/docs/containers/overview>).



**STEP 3** - Create a new folder named "REST API" and open it in VS Code.



**STEP 4** - Open the Command Palette (**Ctrl+Shift+P**) and choose "Python: Create Environment". Select the appropriate app type (Python: Flask) and provide the entry point path for your application.



**STEP 5** – Create a Python file named “`app.py`” inside a `venv` folder.

The screenshot shows the Visual Studio Code interface. The top navigation bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The title bar says "REST API". The left sidebar has icons for Explorer, Search, Find, Open, Recent, and Settings. The Explorer view shows a project structure with a .venv folder containing app.py, .gitignore, and pyvenv.cfg. The Editor pane shows "app.py" with the code ".venv > app.py" and a cursor at line 1. The Terminal pane at the bottom is empty. The status bar at the bottom shows "In 1 Col 1 Spaces 4 UTE-8 CR LF Python 3.10.11 C:\venv\venv" and "Go Live".

**STEP 6** - Write a **Python Flask** code for providing a localhost for the application and also define route to ‘**home**’ using “**GET Methods**”

The screenshot shows the Visual Studio Code (VS Code) interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** REST API
- Explorer View:** Shows a file tree for a project named "REST API". The tree includes ".venv", ".venv\Scripts", ".gitignore", "app.py", and "pyenv.cfg". The "app.py" file is currently selected.
- Code Editor:** Displays the content of "app.py":

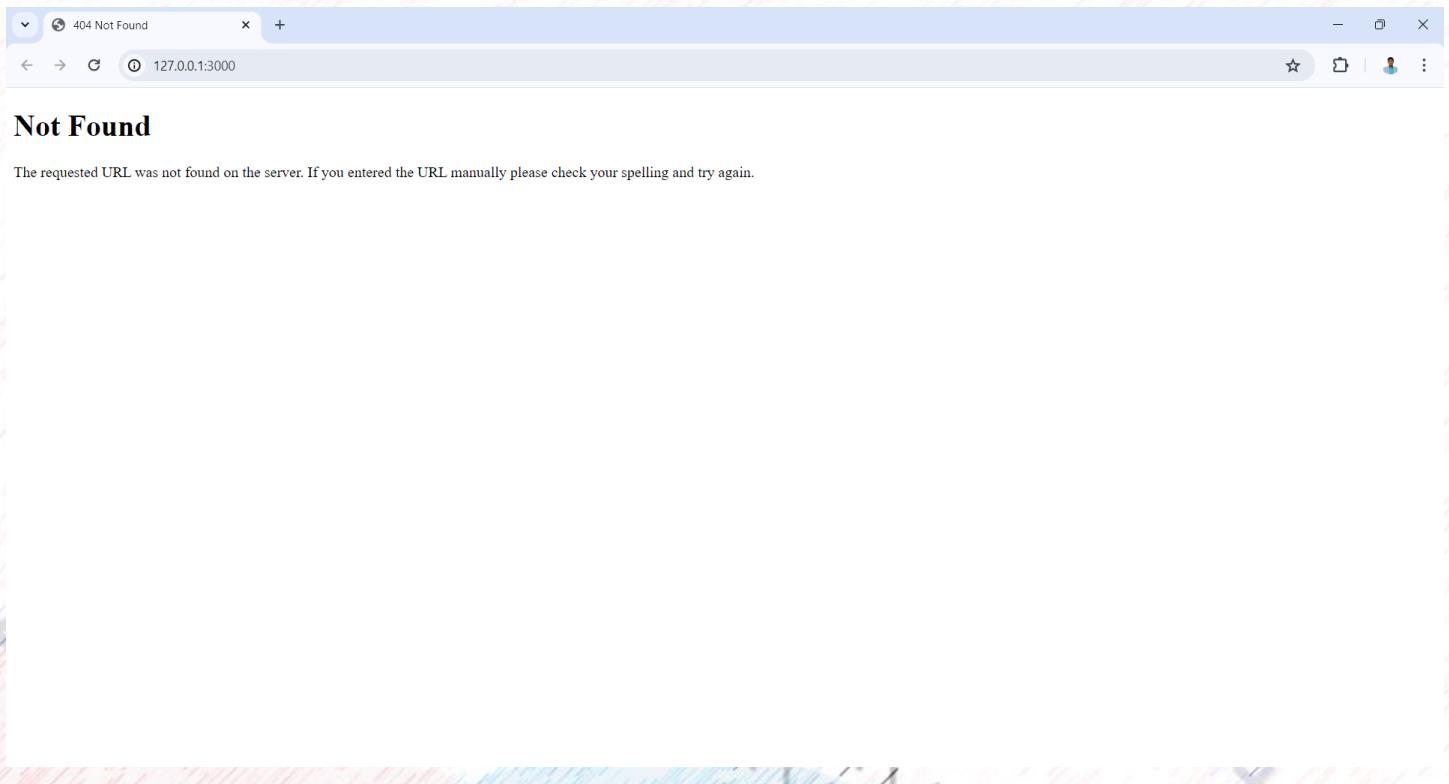
```
from flask import Flask
app=Flask(__name__)
#This is our home page
@app.route('/home',methods=['GET'])
def home_page():
    return "This is my home page"
if __name__ == '__main__':
    app.run(debug=True,port=3000)
```

- Bottom Status Bar:** In 11, Col 34, Spaces 4, UTF-8, CR/LF, Python 3.10.11 (.venv\venv), Go Live, etc.

**STEP 7** – Run the file to your localhost and use “<http://127.0.0.1:3000/home>” to redirect on your home page that you created in code.



**STEP 8** - At the execution time of code you will see HTTP error which is called “[404](#)” error in which “[NOT FOUND](#)” will be displayed. Which defines that you will not decide any route to this location. The 404 code means that a server could not find a client-requested webpage.



## STEP 9 -Open Terminal

and get inside the `venv` folder. Run the command

(`pip freeze > requirements.txt`) to create `requirements.txt` file which contains all python packages for your application.

The screenshot shows the VS Code interface. The terminal at the top right shows the command history:

```
PS C:\Users\hp\OneDrive\Desktop\REST API> cd .venv  
PS C:\Users\hp\OneDrive\Desktop\REST API\.venv> pip freeze > requirements.txt  
PS C:\Users\hp\OneDrive\Desktop\REST API\.venv>
```

The Explorer sidebar shows a project structure under 'REST API' with files like `app.py`, `requirements.txt`, and `pyenv.cfg`. The 'requirements.txt' file is open in the editor, displaying the following content:

```
blinker==1.8.2  
click==8.1.7  
colorama==0.4.6  
Flask==3.0.3  
itsdangerous==2.2.0  
Jinja2==3.1.4  
MarkupSafe==2.1.5  
Werkzeug==3.0.3
```

The status bar at the bottom indicates the file has 1 line, 4 spaces, is in UTF-16 LE encoding, and uses CRLF line endings.

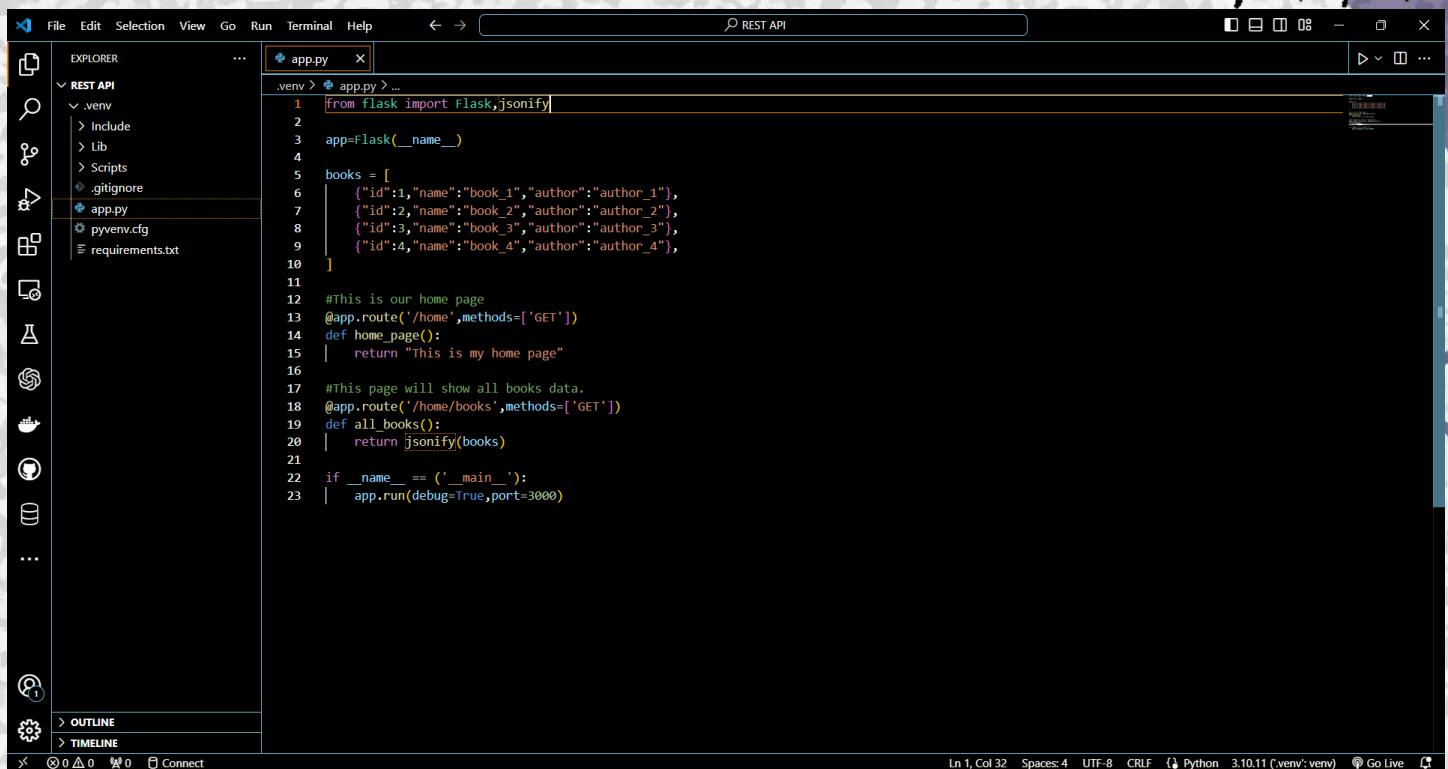
## STEP 10 - Create a json file named "books" and store 4 books data.

The screenshot shows the VS Code interface. The Explorer sidebar shows a project structure under 'REST API' with files like `app.py`, `requirements.txt`, and `pyenv.cfg`. The 'app.py' file is open in the editor, displaying the following code:

```
from flask import Flask  
app=flask(__name__)  
  
books = [  
    {"id":1,"name":"book_1","author":"author_1"},  
    {"id":2,"name":"book_2","author":"author_2"},  
    {"id":3,"name":"book_3","author":"author_3"},  
    {"id":4,"name":"book_4","author":"author_4"},  
]  
  
#This is our home page  
@app.route('/home',methods=['GET'])  
def home_page():  
    return "This is my home page"  
  
if __name__ == ('__main__'):  
    app.run(debug=True,port=3000)
```

The status bar at the bottom indicates the file has 18 lines, 34 spaces, is in UTF-8 encoding, and uses CRLF line endings, and is using Python 3.10.11 (.venv: venv).

**STEP 11 -** Import jsonify function (It sets the correct response headers and content type for JSON responses) and create a new app route named “/books” for displaying books data using ‘GET’ methods.

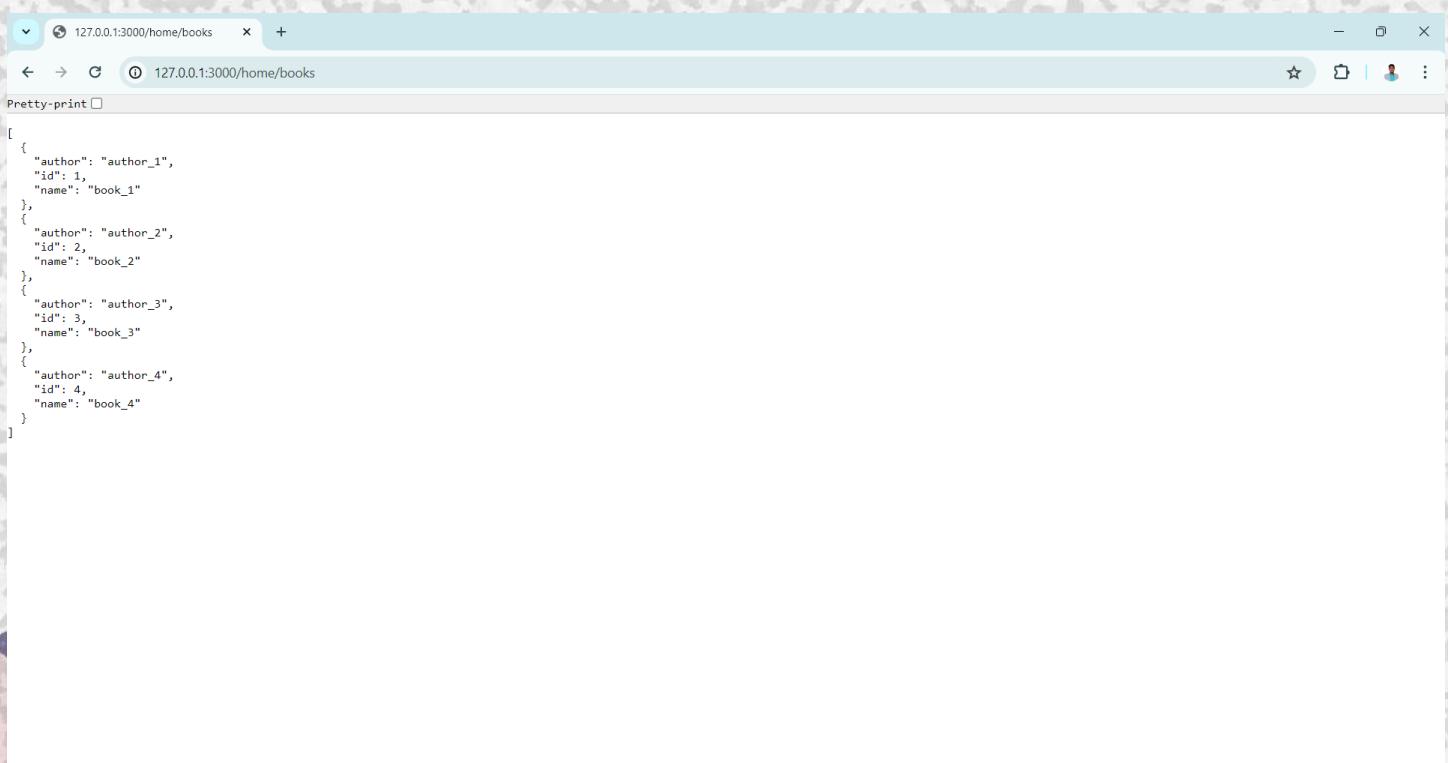


The screenshot shows a code editor interface with a dark theme. On the left is an Explorer sidebar showing a project structure with files like .venv, app.py, pyenv.cfg, and requirements.txt. The main editor area contains the following Python code:

```
from flask import Flask, jsonify
app=Flask(__name__)
books = [
    {"id":1,"name":"book_1","author":"author_1"},
    {"id":2,"name":"book_2","author":"author_2"},
    {"id":3,"name":"book_3","author":"author_3"},
    {"id":4,"name":"book_4","author":"author_4"}
]
#This is our home page
@app.route('/home',methods=['GET'])
def home_page():
    return "This is my home page"
#This page will show all books data,
@app.route('/home/books',methods=['GET'])
def all_books():
    return jsonify(books)
if __name__ == ('__main__'):
    app.run(debug=True,port=3000)
```

At the bottom of the editor, status bar details include: Ln 1, Col 32, Spaces: 4, UTF-8, CRLF, Python 3.10.11 (.venv: venv), Go Live.

**STEP 12 -** Run the code and after opening the route of “/home/books” you will see that all books data which are stored in your file that all should be displayed.



The screenshot shows a browser window with the URL 127.0.0.1:3000/home/books. The page displays a JSON array of book data:

```
[{"author": "author_1", "id": 1, "name": "book_1"}, {"author": "author_2", "id": 2, "name": "book_2"}, {"author": "author_3", "id": 3, "name": "book_3"}, {"author": "author_4", "id": 4, "name": "book_4"}]
```

**STEP 13** - Copy the URL (<http://127.0.0.1:3000/home/books>) and paste it in Postman with "GET" method, and you will see that all the books data has been shown here.

The screenshot shows the Postman application interface. On the left, the sidebar includes 'My Workspace' with 'Collections' (cloudant, demo), 'Environments', and 'History'. The main workspace shows a 'GET new' request to 'demo / new'. The URL is set to `http://127.0.0.1:3000/home/books`. The 'Headers' tab contains six entries. The 'Body' tab displays a JSON response with four items, each representing a book with fields: author, id, and name. The status bar at the bottom indicates the response was successful (Status: 200 OK, Time: 16 ms, Size: 446 B).

```
[{"id": 1, "name": "book_1", "author": "author_1"}, {"id": 2, "name": "book_2", "author": "author_2"}, {"id": 3, "name": "book_3", "author": "author_3"}, {"id": 4, "name": "book_4", "author": "author_4"}]
```

**STEP 14** – After getting the request in Postman, we need to create a new app route (`/home/books/1`) named “`/books/1`” for getting extracting single book data and we use message for invalid book id.

The screenshot shows a Visual Studio Code (VS Code) interface with the following details:

- File Explorer (Left):** Shows the project structure with files: `app.py`, `.venv`, `.env`, `requirements.txt`, and `pyenv.cfg`.
- Code Editor (Center):** Displays the content of `app.py` which contains a Flask application for a REST API. The code includes routes for home, all books, and a specific book by ID.
- Terminal (Bottom):** Shows the command `REST API` entered in the terminal bar.
- Status Bar (Bottom):** Shows the status "In 28. Col 53" and other system information.

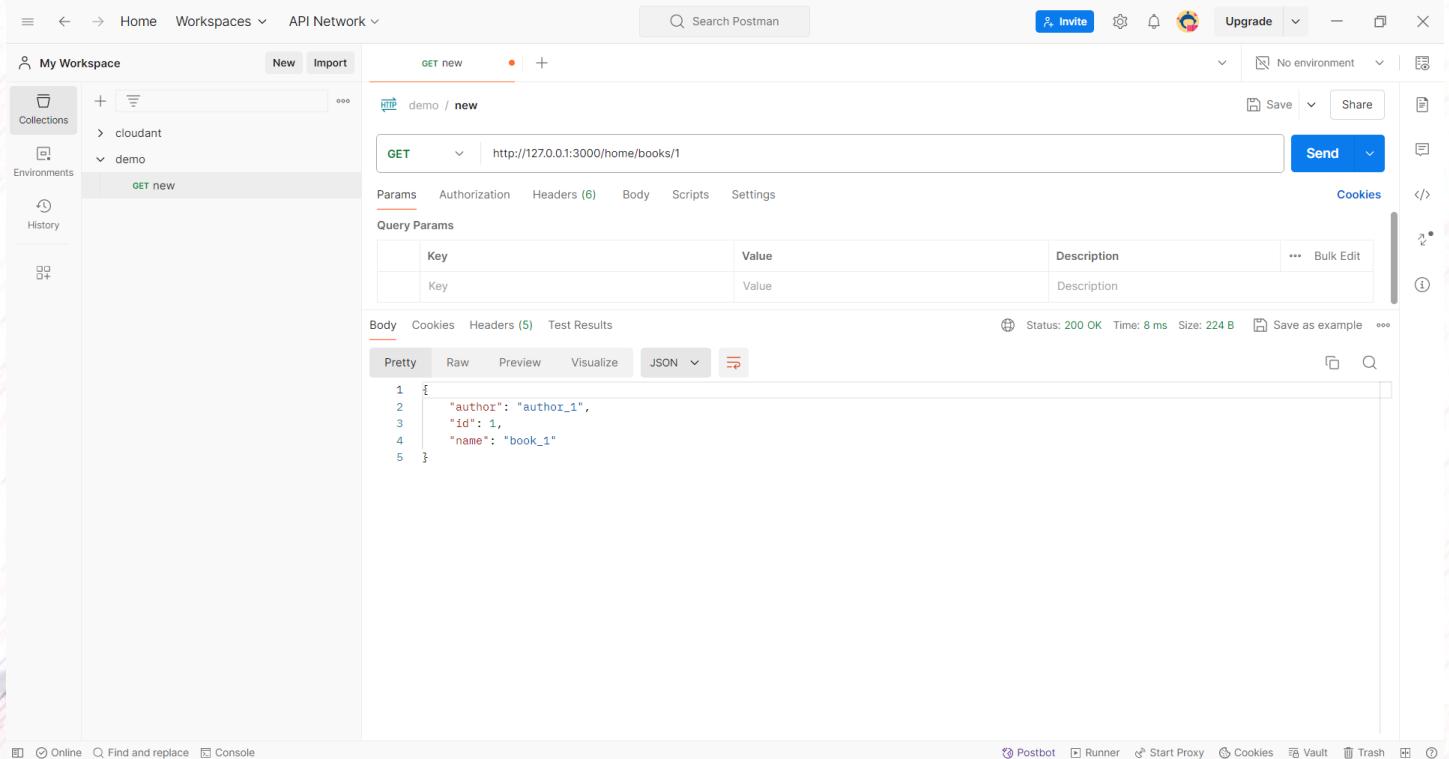
**STEP 15** - Run the code and after opening the route of “[/home/books/1](http://127.0.0.1:3000/home/books/1)” you will see that single books data which are stored in your file that all should be displayed.



A screenshot of a web browser window. The address bar shows the URL [127.0.0.1:3000/home/books/1](http://127.0.0.1:3000/home/books/1). The page content displays a JSON object:

```
{ "author": "author_1", "id": 1, "name": "book_1" }
```

**STEP 16** - Copy the [URL \(<http://127.0.0.1:3000/home/books/1>\)](http://127.0.0.1:3000/home/books/1) and paste it in [Postman](#) with “[GET](#)” method, and you will see that the single books data has been shown here.



A screenshot of the Postman application interface. On the left, the sidebar shows “My Workspace” with “Collections”, “Environments”, and “History”. The main area shows a “GET new” request under the “demo / new” collection. The request URL is <http://127.0.0.1:3000/home/books/1>. The “Body” tab is selected, showing the JSON response:

```
1 {  
2   "author": "author_1",  
3   "id": 1,  
4   "name": "book_1"  
5 }
```

**STEP 17** - After getting the output in Postman, we need to import request function (It sends seven main kinds of request to a web server) and create a structure for adding new book data from external.

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows a project structure with files: .venv, app.py, pyvenv.cfg, and requirements.txt.
- Editor:** The active file is `app.py`, containing Python code for a REST API. The code defines routes for home, books, and book\_id, and includes a POST method for adding new books. It uses the Flask framework and JSONify for responses.
- Status Bar:** Shows the current file is `app.py`, line 41, column 29. It also shows settings for Spaces: 4, UTF-8, CRLF, Python 3.10.11 (.venv:venv), and Go Live.

**STEP 18** - Run the code at localhost and copy the URL and in this we add a new book data ("author": "author\_5","id": 5, "name":"book\_5") with book id "/books/5" using 'POST' methods in Postman.

The screenshot shows the Postman application interface. The left sidebar includes 'My Workspace' with 'Collections' (cloudant, demo), 'Environments', and 'History'. The main workspace shows a 'POST new' collection with a 'demo / new' item. The request details are as follows:

- Method:** POST
- URL:** <http://127.0.0.1:3000/home/books>
- Body:** JSON (selected)
- Body Content:**

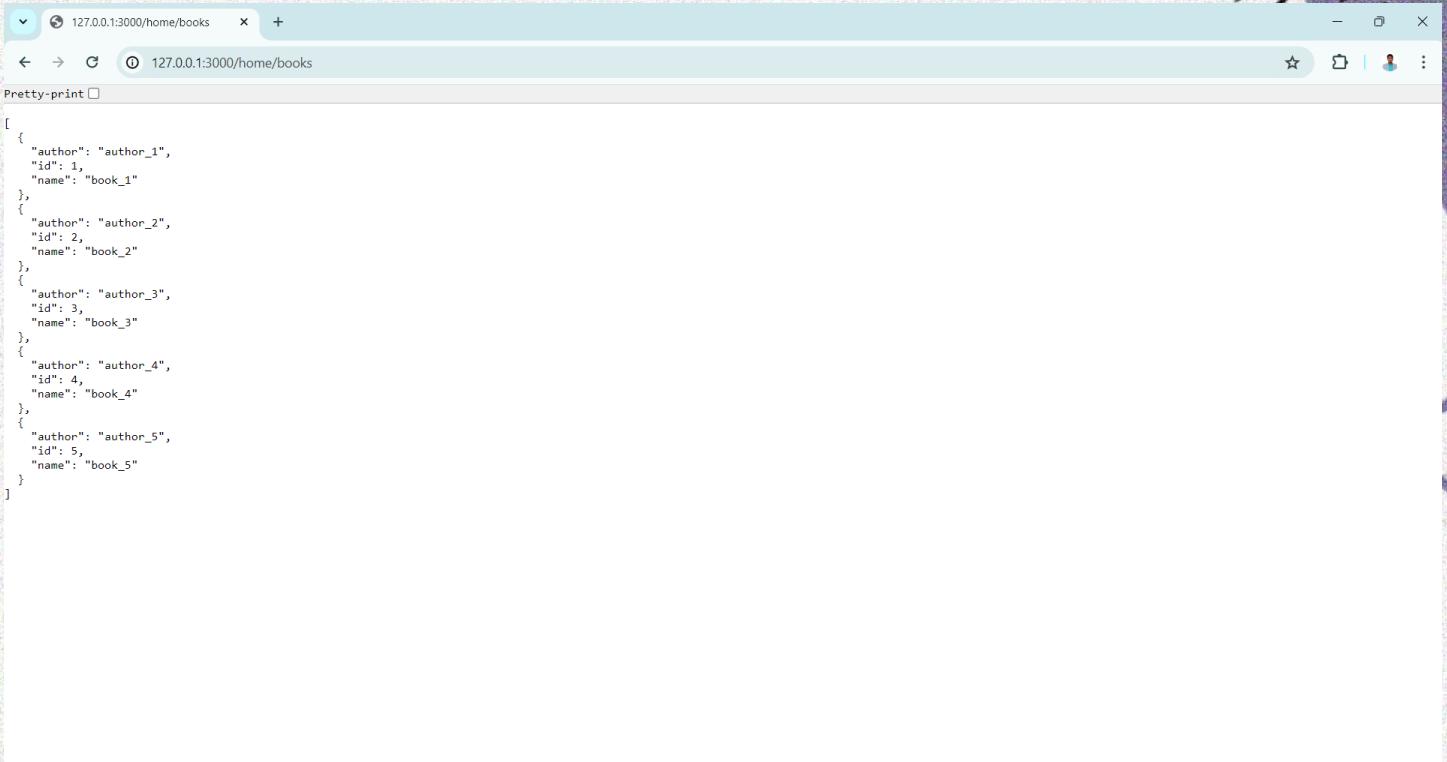
```
1 {  
2   ... "id":5,  
3   ... "name":"book_5",  
4   ... "author":"author_5"  
5 }
```

The response details are as follows:

- Status:** 200 OK
- Time:** 10 ms
- Size:** 209 B
- Body:** JSON (Pretty, Raw, Preview, Visualize, JSON dropdown set to JSON)
- Response Content:**

```
1 {  
2   ... "message": "Book added successfully"  
3 }
```

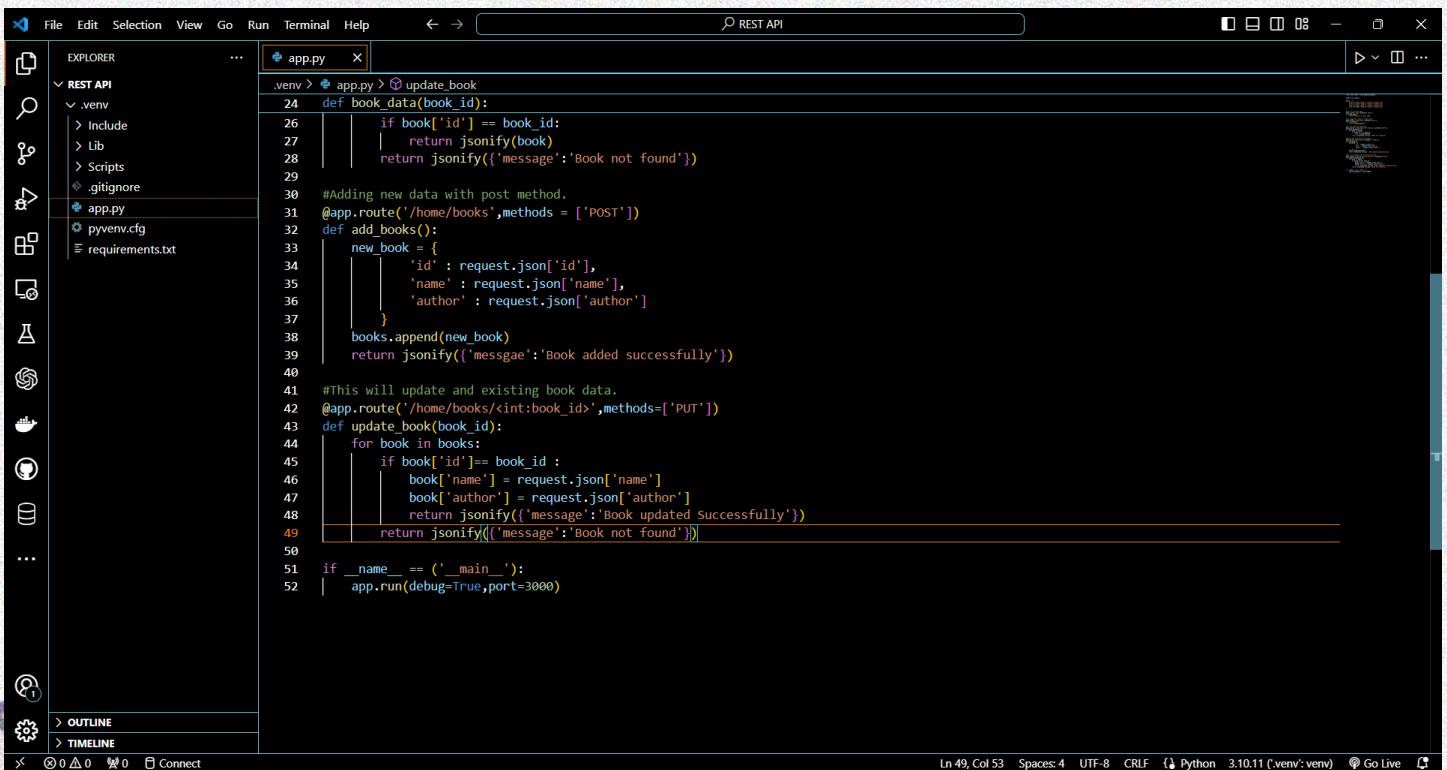
**STEP 19** - After running the code the message prints that “Book added successfully”, Let’s check the update in localhost by refreshing the link (You will see that new book data added successfully)



A screenshot of a web browser window showing the URL `127.0.0.1:3000/home/books`. The page displays a JSON array of five book entries:

```
[{"author": "author_1", "id": 1, "name": "book_1"}, {"author": "author_2", "id": 2, "name": "book_2"}, {"author": "author_3", "id": 3, "name": "book_3"}, {"author": "author_4", "id": 4, "name": "book_4"}, {"author": "author_5", "id": 5, "name": "book_5"}]
```

**STEP 20** – For updating an existing book data we use “PUT” method. Create a new structure for using PUT methods by giving the condition of matching Book id of particular book data.



A screenshot of a code editor showing the `app.py` file. The code defines two routes: a POST route for adding new books and a PUT route for updating existing books by their ID. The PUT route checks if the provided book ID matches the book's ID in the database, updates the book's name and author, and returns a success message. If the book is not found, it returns a message indicating the book was not found.

```
def book_data(book_id):
    if book['id'] == book_id:
        return jsonify(book)
    return jsonify({'message': 'Book not found'})

#Adding new data with post method.
@app.route('/home/books',methods = ['POST'])
def add_books():
    new_book = {
        'id' : request.json['id'],
        'name' : request.json['name'],
        'author' : request.json['author']
    }
    books.append(new_book)
    return jsonify({'messgae': 'Book added successfully'})

#This will update and existing book data.
@app.route('/home/books/<int:book_id>',methods=[ 'PUT'])
def update_book(book_id):
    for book in books:
        if book['id']== book_id :
            book['name'] = request.json['name']
            book['author'] = request.json['author']
            return jsonify({'message':'Book updated Successfully'})
    return jsonify([{'message': 'Book not found'}])

if __name__ == ('__main__'):
    app.run(debug=True,port=3000)
```

**STEP 21** - Run the code and select the book which you want to change or update and copy the **URL**. Paste it in Postman with "PUT" method and change the data of selected book.

The screenshot shows the Postman interface. In the left sidebar, there's a 'My Workspace' section with a 'Collections' tab selected, showing a 'demo' collection. The main workspace has a 'PUT new' tab open, with the URL set to `http://127.0.0.1:3000/home/books/1`. The 'Body' tab is selected, showing the following JSON payload:

```

1 {
2   ...
3   "id":1,
4   ...
5   "name":"bookname1",
6   ...
7   "author":"authorname1"
8 }

```

Below the body, the response tab shows a successful `200 OK` status with the message `"Book updated Successfully"`.

**STEP 22** - For deleting an existing book data we use "**DELETE**" method. Create a new structure for using DELETE method by giving the condition of matching Book id of particular book data and by using **remove** function we delete the existing book data.

The screenshot shows the VS Code editor with the file `app.py` open. The code defines three main functions: `add_books`, `update_book`, and `delete_book`.

```

1 # This will add a new book data.
2 @app.route('/home/books', methods=['POST'])
3 def add_books():
4     new_book = {
5         'name' : request.json['name'],
6         'author' : request.json['author']
7     }
8     books.append(new_book)
9     return jsonify({'message': 'Book added successfully'})
10
11 # This will update an existing book data.
12 @app.route('/home/books/<int:book_id>', methods=['PUT'])
13 def update_book(book_id):
14     for book in books:
15         if book['id'] == book_id:
16             book['name'] = request.json['name']
17             book['author'] = request.json['author']
18             return jsonify({'message': 'Book updated Successfully'})
19         return jsonify({'message': 'Book not found'})
20
21 # This will delete an existing book data.
22 @app.route('/home/books/<int:book_id>', methods=['DELETE'])
23 def delete_book(book_id):
24     for book in books:
25         if book['id'] == book_id:
26             books.remove(book)
27             return jsonify({'message': 'Book deleted Successfully'})
28         return jsonify({'message': 'Book not found'})
29
30 if __name__ == ('__main__'):
31     app.run(debug=True, port=3000)

```

**STEP 23** - Run the code and select the book which you want to delete and copy the URL. Paste it in Postman with “DELETE” method and provide the book id to it.

The screenshot shows the Postman interface. In the left sidebar, there's a 'Collections' section with 'cloudant' and 'demo' listed, and an 'Environments' section. The main workspace has a 'demo / new' collection expanded. A 'DELETE' request is selected with the URL `http://127.0.0.1:3000/home/books/1`. The 'Body' tab is active, showing a JSON payload with `{ "id": 1 }`. The 'Headers' tab shows '(8)' headers. The 'Params' tab shows no parameters. The 'Body' tab also includes options for 'none', 'form-data', 'x-www-form-urlencoded', 'raw', 'binary', 'GraphQL', and 'JSON'. The 'JSON' option is selected. Below the body, the response status is 200 OK with the message "Book deleted Successfully". The 'Pretty' tab is selected in the preview area.

**STEP 24** - Your book has been deleted successfully.

The screenshot shows a browser window with the URL `127.0.0.1:3000/home/books`. The page displays a JSON array of books:

```
[{"author": "author_2", "id": 2, "name": "book_2"}, {"author": "author_3", "id": 3, "name": "book_3"}, {"author": "author_4", "id": 4, "name": "book_4"}]
```

**GIT HUB REPOSITORY LINK :**

["https://github.com/shantanupandey22/HTTPS-methods/"](https://github.com/shantanupandey22/HTTPS-methods/)