**1. Architectural Principles of REST (Representational State Transfer)**

REST is an **architectural style** for designing networked applications. It relies on a **stateless, client-server, cacheable communications protocol** — the HTTP protocol is almost always used.

**Key Principles:**

| **Principle** | **Explanation** |
| --- | --- |
| **Stateless** | Each request from the client to the server must contain all the information needed to understand and process the request. The server does not store session state between requests. |
| **Client-Server** | The client and server are separate entities. This separation allows each to evolve independently. |
| **Cacheable** | Responses should be defined as cacheable or non-cacheable to improve performance. |
| **Uniform Interface** | A consistent and standard way of interacting with resources. This simplifies architecture and improves visibility. |
| **Layered System** | A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary (proxy, gateway, etc.). |
| **Code on Demand (Optional)** | Servers can temporarily extend or customize the functionality of a client by transferring executable code (e.g., JavaScript). |

**2. Key Components of a REST API**

**a. Resources**

* These are the key abstractions in REST.
* Anything that can be named and represented digitally (e.g., Users, Books, Orders).
* Represented typically in JSON or XML format.

**Example**:  
A book resource might look like:

{

"id": 1,

"title": "1984",

"author": "George Orwell"

}

**b. Endpoints**

* URLs used to access resources.

**Example**:  
GET /books → Fetch list of books  
GET /books/1 → Fetch book with ID 1  
POST /books → Create a new book

**c. HTTP Methods**

* Standard HTTP methods are used to perform CRUD operations (see section 4 below).

**d. Status Codes**

* HTTP status codes indicate the result of a request.

| **Code** | **Meaning** |
| --- | --- |
| 200 | OK - Request succeeded |
| 201 | Created - Resource successfully created |
| 400 | Bad Request - Client error |
| 401 | Unauthorized - Auth needed |
| 404 | Not Found - Resource doesn’t exist |
| 500 | Internal Server Error |

**3. REST vs Other API Architectures**

| **Feature** | **REST** | **SOAP** | **GraphQL** |
| --- | --- | --- | --- |
| Protocol | HTTP | HTTP, SMTP, TCP | HTTP |
| Data Format | JSON (mainly), XML | XML | JSON |
| Flexibility | Fixed endpoints, structured | Rigid, highly structured | Flexible queries |
| Performance | Lightweight | Heavy | Efficient (fetch only needed data) |
| Error Handling | HTTP status codes | SOAP Fault messages | Custom error objects |
| Use Cases | Web services, mobile APIs | Enterprise-level transactions | Modern front-end/back-end APIs |

**4. HTTP Methods in RESTful Services**

**a. GET**

* **Purpose**: Retrieve a resource or list of resources.
* **Safe** and **idempotent** (does not change the server state).
* **Use Case**: Viewing a book detail.

**Example**:

GET /books/1

**b. POST**

* **Purpose**: Create a new resource.
* **Not idempotent**.
* **Use Case**: Adding a new book to the library.

**Example**:

POST /books

{

"title": "Things Fall Apart",

"author": "Chinua Achebe"

}

**c. PUT**

* **Purpose**: Update a resource entirely.
* **Idempotent**.
* **Use Case**: Replacing a book’s information.

**Example**:

PUT /books/1

{

"title": "New Title",

"author": "New Author"

}

**d. DELETE**

* **Purpose**: Remove a resource.
* **Idempotent**.
* **Use Case**: Deleting a book.

**Example**:

DELETE /books/1

**e. PATCH**

* **Purpose**: Partially update a resource.
* **Use Case**: Updating just the author of a book.

**Example**:

PATCH /books/1

{

"author": "Updated Author"

}

## ****Security and Best Practices for REST APIs****

### ****1. Common Security Practices****

#### a. **Authentication**

Ensures only authorized users can access your API.

* **Basic Authentication**: Username and password encoded in base64 (not secure unless used with HTTPS).
* **Token-Based Authentication** (e.g., JWT - JSON Web Token): A token is issued after successful login and is used for subsequent requests.
* **OAuth 2.0**: Secure and scalable way to grant access using access tokens. Often used for third-party integrations (e.g., Google, Facebook logins).

#### b. **Authorization**

Controls what authenticated users are allowed to do.

* Example: Admin users can add/delete data, while regular users can only read.
* Use **Role-Based Access Control (RBAC)** to enforce authorization levels.

#### c. **HTTPS**

* All API traffic should be encrypted using HTTPS to protect against data interception and man-in-the-middle attacks.
* Never allow access to your REST API over plain HTTP in production.

#### d. **Input Validation & Sanitization**

* Validate all user inputs to avoid injection attacks (e.g., SQL injection, XSS).
* Sanitize inputs to remove harmful characters.

#### e. **Rate Limiting & Throttling**

* Protect your API from abuse by limiting the number of requests a client can make per time period.
* Example: Allow 100 requests per minute per user/IP.
* Tools: NGINX, API Gateways (e.g., AWS API Gateway), or custom middleware.

#### f. **CORS (Cross-Origin Resource Sharing)**

* Configure CORS policies to prevent unauthorized domains from making API calls.
* Only allow trusted origins.

## Best Practices for Designing and Implementing REST APIs

### 1️ ****API Versioning****

Versioning ensures that when you update your API, you don’t break existing applications that rely on the older version.

#### **Why Version?**

* Prevents breaking changes for existing users
* Allows smooth transition to new features

#### **Best Practices:**

* Use versioning in the **URL**:  
  GET /api/v1/books
* You can also version via **headers**, but URL versioning is more transparent.
* Use semantic versioning (v1, v2) rather than dates unless you release very frequently.

### 2️. ****Error Handling****

Proper error handling helps clients understand what went wrong and how to fix it.

#### **Best Practices:**

* Use standard **HTTP status codes**:
  + 200 OK – Successful request
  + 201 Created – Resource successfully created
  + 400 Bad Request – Invalid input from client
  + 401 Unauthorized – Authentication required
  + 403 Forbidden – Access denied
  + 404 Not Found – Resource doesn't exist
  + 500 Internal Server Error – Server-side issue
* Return a consistent **error format**:

{

"error": "ValidationError",

"message": "The 'title' field is required",

"status": 400

}

* Avoid leaking sensitive information in error messages.

### 3️. ****Rate Limiting****

Rate limiting protects your API from being overwhelmed by too many requests (DDoS attacks or overuse).

#### **Best Practices:**

* Set limits such as:
  + 1000 requests per user per hour
  + 50 requests per IP per minute
* Return proper headers to inform clients of their usage:

X-Rate-Limit-Limit: 100

X-Rate-Limit-Remaining: 10

X-Rate-Limit-Reset: 1692990823

* On limit exceeded, return:

429 Too Many Requests

* Tools for implementation:
  + API Gateways (e.g., **AWS API Gateway**, **Kong**, **NGINX**)
  + Custom middleware in **FastAPI**, **Express.js**, **Django**, etc.

### 4️. ****Use Proper HTTP Methods****

Each HTTP method should reflect its purpose in CRUD:

| **Method** | **Use Case** | **Example** |
| --- | --- | --- |
| GET | Retrieve data | GET /books |
| POST | Create new resource | POST /books |
| PUT | Replace resource | PUT /books/1 |
| PATCH | Update part of resource | PATCH /books/1 |
| DELETE | Remove resource | DELETE /books/1 |

### 5️. ****Pagination, Filtering, and Sorting****

Avoid returning large data sets. Instead:

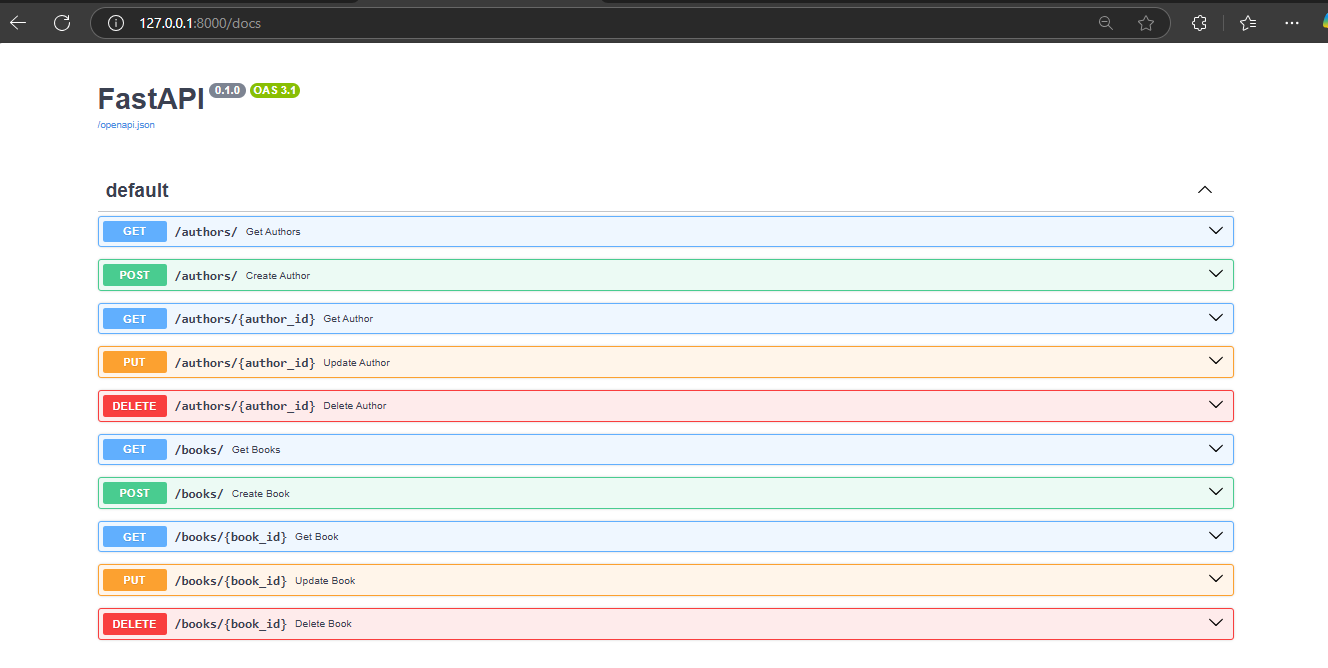
* **Paginate** responses:  
  GET /books?page=1&limit=10
* **Filter** results:  
  GET /books?author=tolkien
* **Sort**:  
  GET /books?sort=published\_date

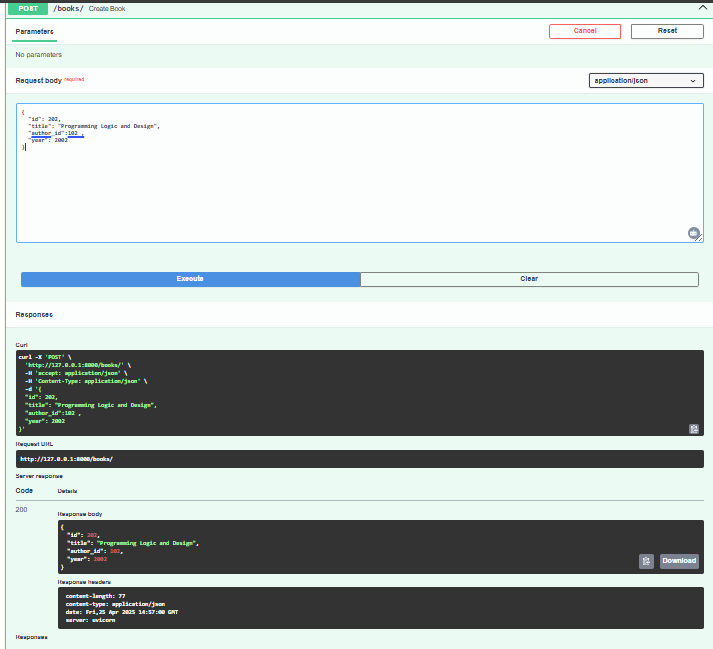
### 6️. ****Consistent Resource Naming****

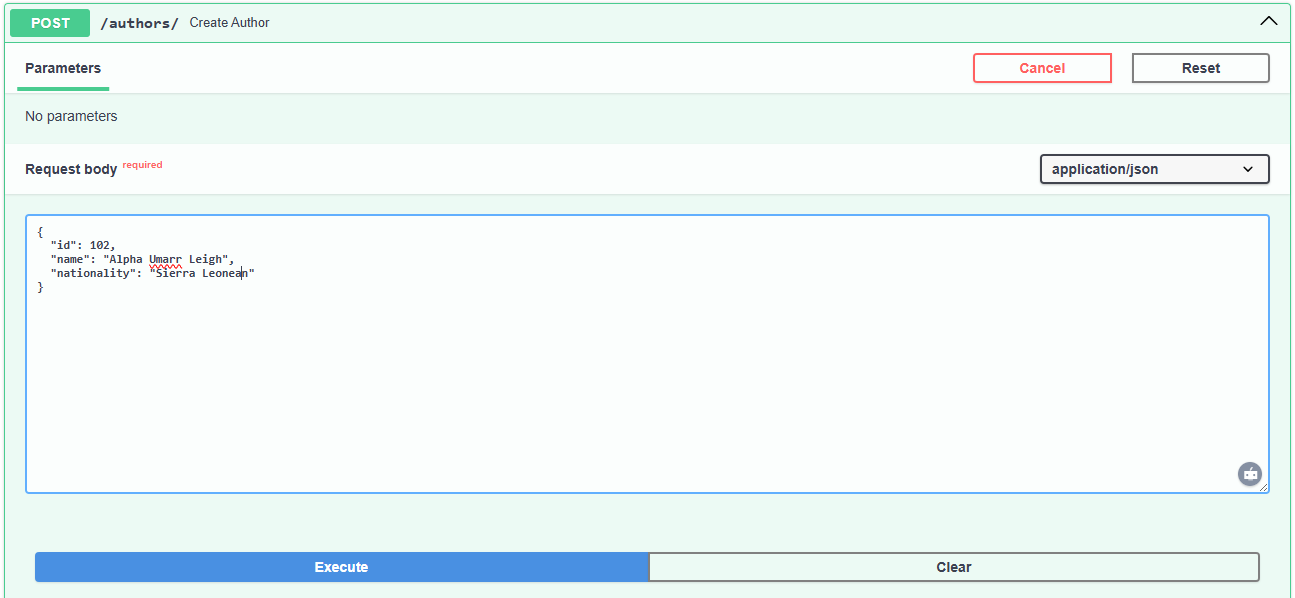
* Use **plural nouns** and **lowercase** for endpoints.  
  GET /books, GET /authors/1/books
* Avoid verbs:  
  GET /getBooks

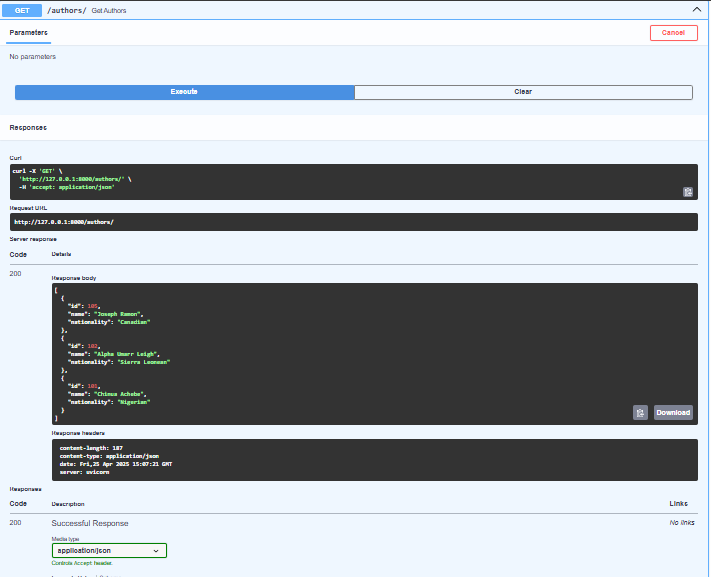
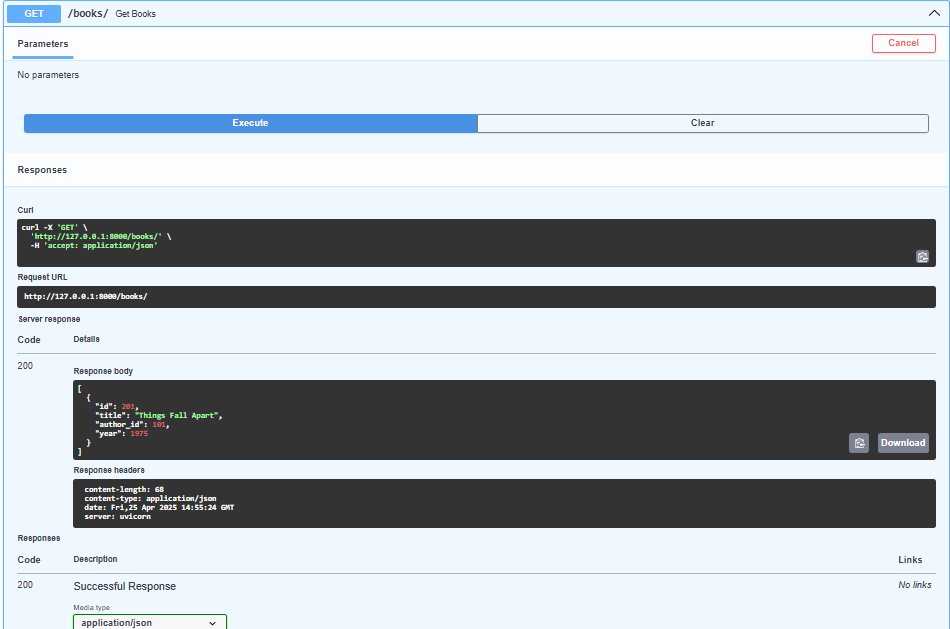
**PRACTICAL IMPLEMENTATION:**

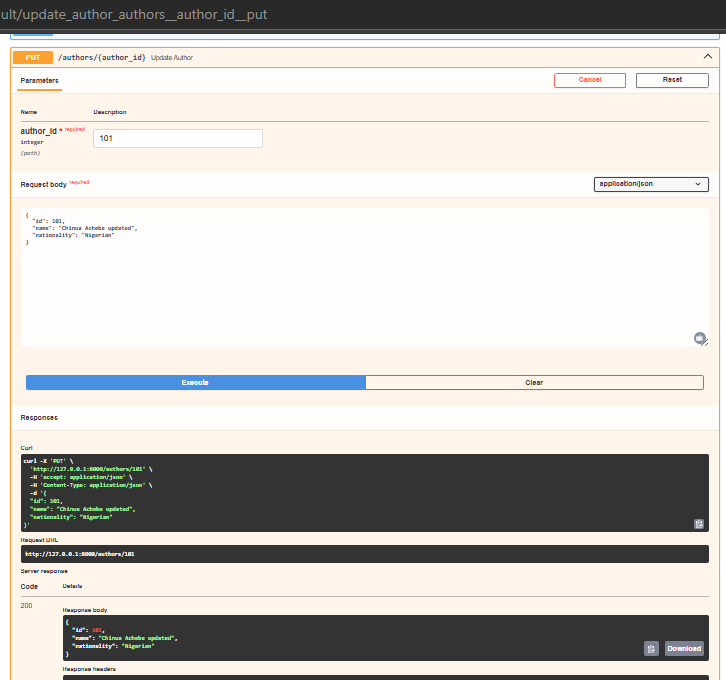
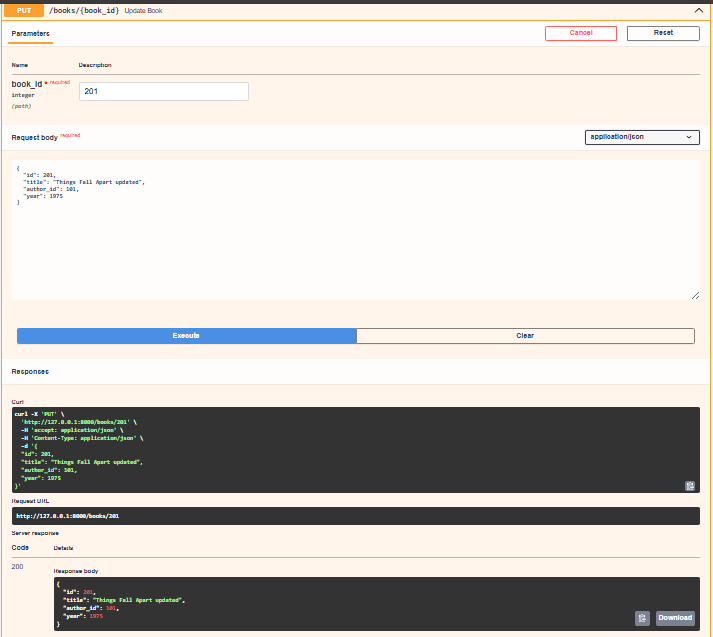
**Design and implement a simple REST API.**

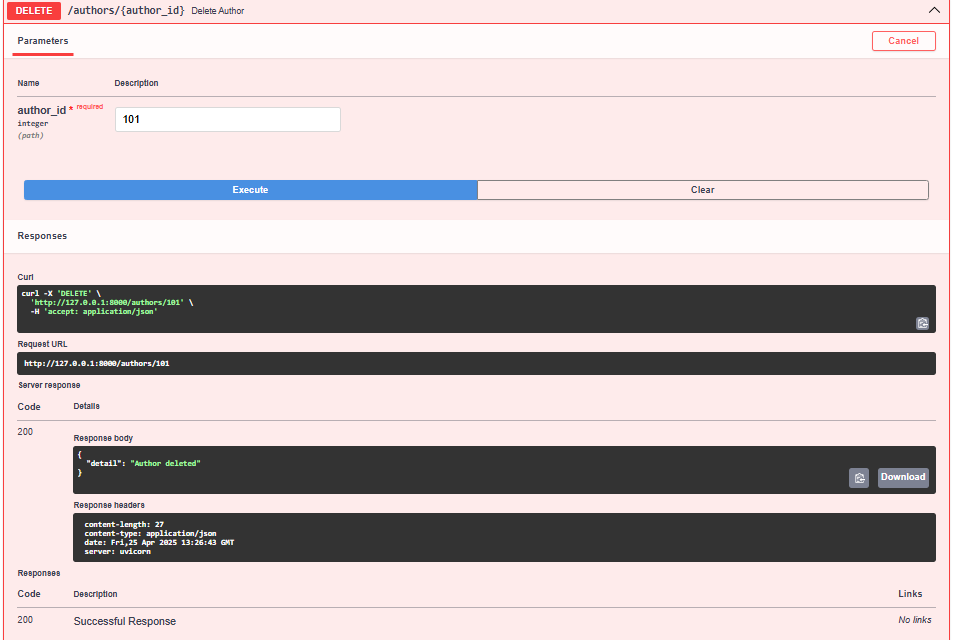
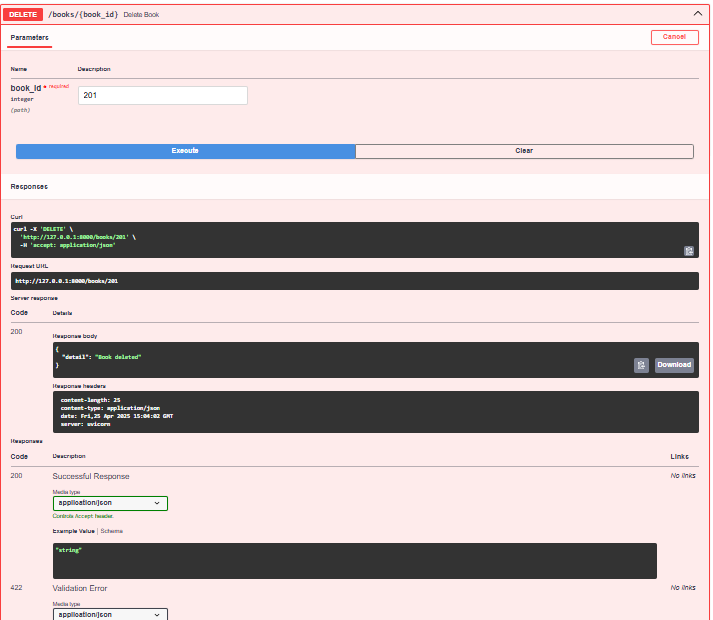
**SWAGGER UI**

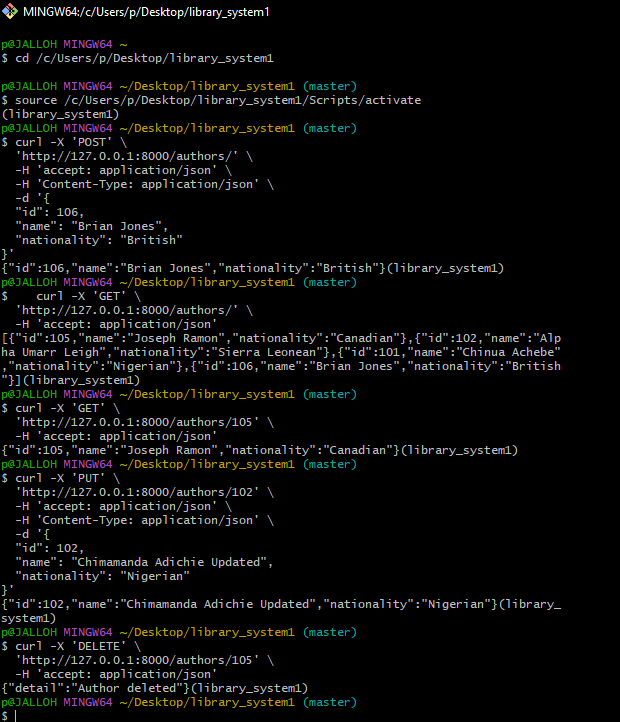
**CRUD OPERATIONS TESTING IMAGES USING SWAGGER UI**

****

****

****

****

**CURL COMMAND TEST**

**MAIN.PY SOURCE CODE**

from fastapi import FastAPI, HTTPException  
from pydantic import BaseModel  
from typing import List, Optional  
  
app = FastAPI()  
  
# In-memory data store  
books\_db = {}  
authors\_db = {}  
  
# Models  
class Author(BaseModel):  
 id: int  
 name: str  
 nationality: Optional[str] = None  
  
class Book(BaseModel):  
 id: int  
 title: str  
 author\_id: int  
 year: Optional[int] = None  
  
# ------------------ AUTHORS ENDPOINTS ------------------ #  
  
@app.post("/authors/", response\_model=Author)  
def create\_author(author: Author):  
 if author.id in authors\_db:  
 raise HTTPException(status\_code=400, detail="Author already exists")  
 authors\_db[author.id] = author  
 return author  
  
@app.get("/authors/", response\_model=List[Author])  
def get\_authors():  
 return list(authors\_db.values())  
  
@app.get("/authors/{author\_id}", response\_model=Author)  
def get\_author(author\_id: int):  
 if author\_id not in authors\_db:  
 raise HTTPException(status\_code=404, detail="Author not found")  
 return authors\_db[author\_id]  
  
@app.put("/authors/{author\_id}", response\_model=Author)  
def update\_author(author\_id: int, author: Author):  
 if author\_id != author.id or author\_id not in authors\_db:  
 raise HTTPException(status\_code=404, detail="Author not found")  
 authors\_db[author\_id] = author  
 return author  
  
@app.delete("/authors/{author\_id}")  
def delete\_author(author\_id: int):  
 if author\_id not in authors\_db:  
 raise HTTPException(status\_code=404, detail="Author not found")  
 del authors\_db[author\_id]  
 return {"detail": "Author deleted"}  
  
# ------------------ BOOKS ENDPOINTS ------------------ #  
  
@app.post("/books/", response\_model=Book)  
def create\_book(book: Book):  
 if book.id in books\_db:  
 raise HTTPException(status\_code=400, detail="Book already exists")  
 if book.author\_id not in authors\_db:  
 raise HTTPException(status\_code=404, detail="Author does not exist")  
 books\_db[book.id] = book  
 return book  
  
@app.get("/books/", response\_model=List[Book])  
def get\_books():  
 return list(books\_db.values())  
  
@app.get("/books/{book\_id}", response\_model=Book)  
def get\_book(book\_id: int):  
 if book\_id not in books\_db:  
 raise HTTPException(status\_code=404, detail="Book not found")  
 return books\_db[book\_id]  
  
@app.put("/books/{book\_id}", response\_model=Book)  
def update\_book(book\_id: int, book: Book):  
 if book\_id != book.id or book\_id not in books\_db:  
 raise HTTPException(status\_code=404, detail="Book not found")  
 books\_db[book\_id] = book  
 return book  
  
@app.delete("/books/{book\_id}")  
def delete\_book(book\_id: int):  
 if book\_id not in books\_db:  
 raise HTTPException(status\_code=404, detail="Book not found")  
 del books\_db[book\_id]  
 return {"detail": "Book deleted"}