

FSD ASSIGNMENT - 6

By-Yashvardhan Tekavade
PB-15 Batch-1
Panel-1 TY-CSF

Aim: Develop a set of REST APIs using Express and Node.

Objectives:

1. To define HTTP GET and POST operations.
2. To understand and make use of 'REST', 'a REST endpoint', 'API Integration', and 'API Invocation'
3. To understand the use of a REST Client to make POST and GET requests to an API.

Theory

1. What is REST API?

REST API, or Representational State Transfer Application Programming Interface, is an architectural style for designing networked applications. It was introduced by Roy Fielding in 2000. RESTful APIs adhere to a set of principles and constraints that promote a scalable, stateless, and standardized approach to communication between client and server over the Internet.

Key principles of RESTful APIs include statelessness, resource-based identification using URIs (Uniform Resource Identifiers), representation of resources in agreed-upon formats (such as JSON or XML), a uniform and consistent interface, and client-server separation.

RESTful APIs use standard HTTP methods (GET, POST, PUT, PATCH, DELETE) to perform operations on resources, making them widely used for building web services. They offer simplicity, scalability, and ease of integration, making them a common choice in modern web development.

2. The main purpose of REST API.

REST API's main purpose is to enable standardized and efficient communication between diverse software systems over the Internet. It achieves this through simplicity, scalability, and flexibility, utilizing standard HTTP methods, a stateless approach, and a uniform interface. REST APIs facilitate resource management and CRUD operations while promoting ease of integration in web development and diverse application environments.

REST APIs serve key purposes:

Interoperability: Enables seamless communication between diverse systems and platforms, supporting applications in different languages and devices.

Simplicity: Designed with a straightforward approach, using standard HTTP methods and a uniform interface for easy understanding and implementation by developers.

Scalability: Stateless design allows servers to handle numerous requests without storing client states between them, promoting scalability.

Flexibility: Adaptable to various data formats (e.g., JSON or XML), accommodating diverse client and server requirements.

Resource Management: Manages resources through standard CRUD operations, enhancing data manipulation efficiency.

Statelessness: Simplifies communication by eliminating the need for servers to store client state between requests.

Uniform Interface: Adheres to a consistent interface, improving the user experience and simplifying API development and consumption.

Ease of Integration: Widely used in web development for seamless integration, leveraging existing HTTP infrastructure and allowing clients to consume APIs with standard methods.

FAQs

1. What are HTTP Request types?

HTTP (Hypertext Transfer Protocol) supports several request methods or types, each indicating the desired action to be performed on a resource. These methods are part of the HTTP protocol and are used by clients to communicate with servers. The most common HTTP request types include:

- GET Retrieves data.
- POST: Submits data for processing.
- PUT: Updates or creates a resource.
- PATCH: Partially updates a resource.
- DELETE: Removes a resource.
- OPTIONS: Describes communication options.
- HEAD: Retrieves headers without a body.
- TRACE: Echoes the received request.
- CONNECT: Establishes a tunnel to a server.

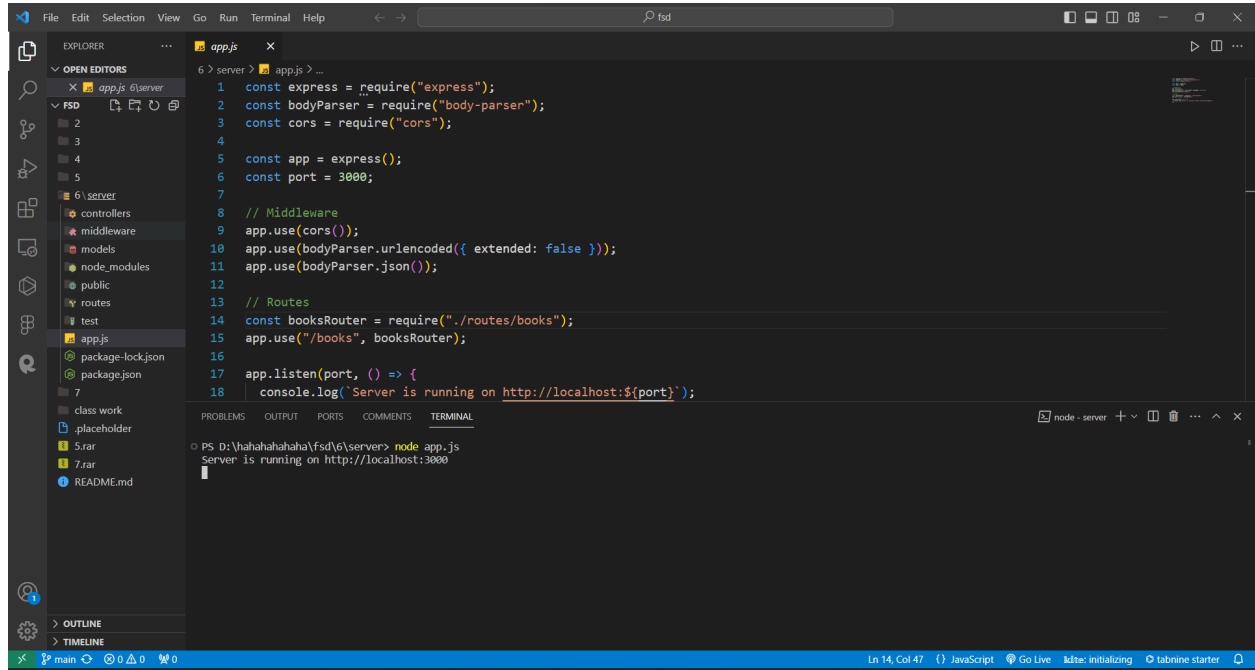
These methods enable standardized client-server communication in web development and API design.

Sample Problem Statements:

Creating and adding new book records in the book database using REST API.

Output: Screenshots of the output to be attached.

1) Start Server

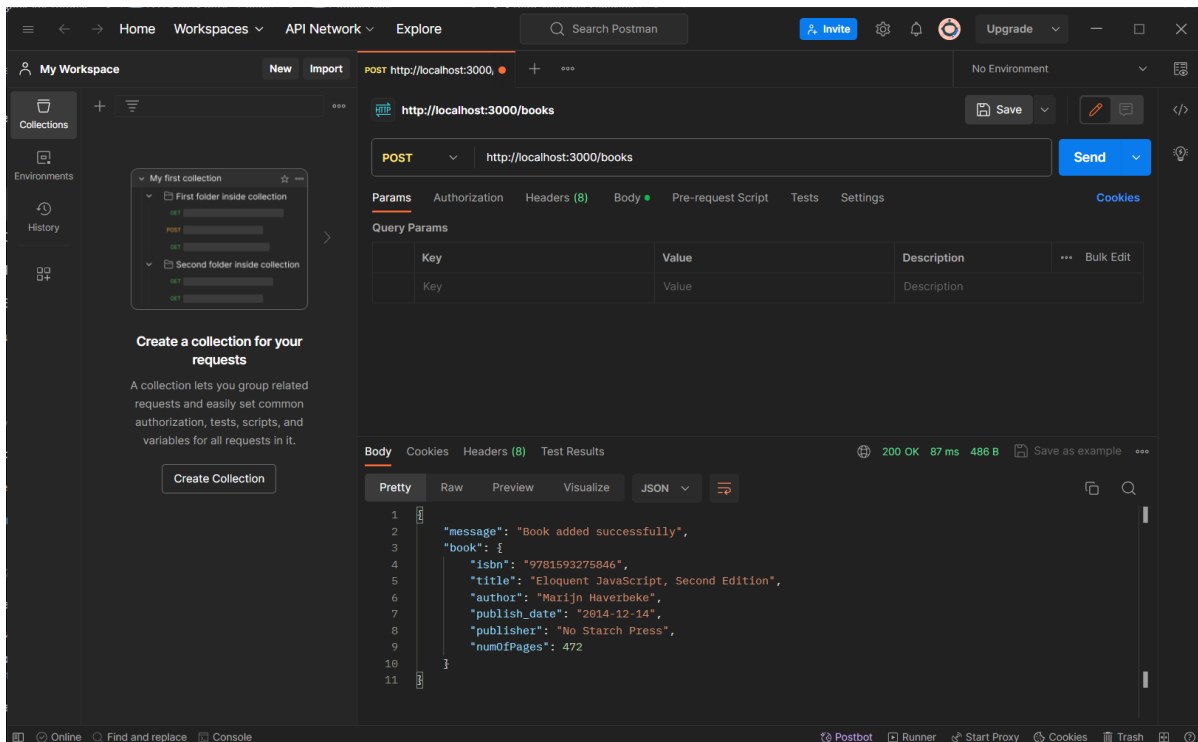


The screenshot shows the Visual Studio Code editor with a project named 'app.js' open. The file explorer on the left shows the project structure, including 'server', 'controllers', 'middleware', 'models', 'node_modules', 'public', 'routes', 'test', 'package-lock.json', 'package.json', 'class work', 'placeholder', '5.rar', '7.rar', and 'README.md'. The main editor displays the 'app.js' file with the following code:

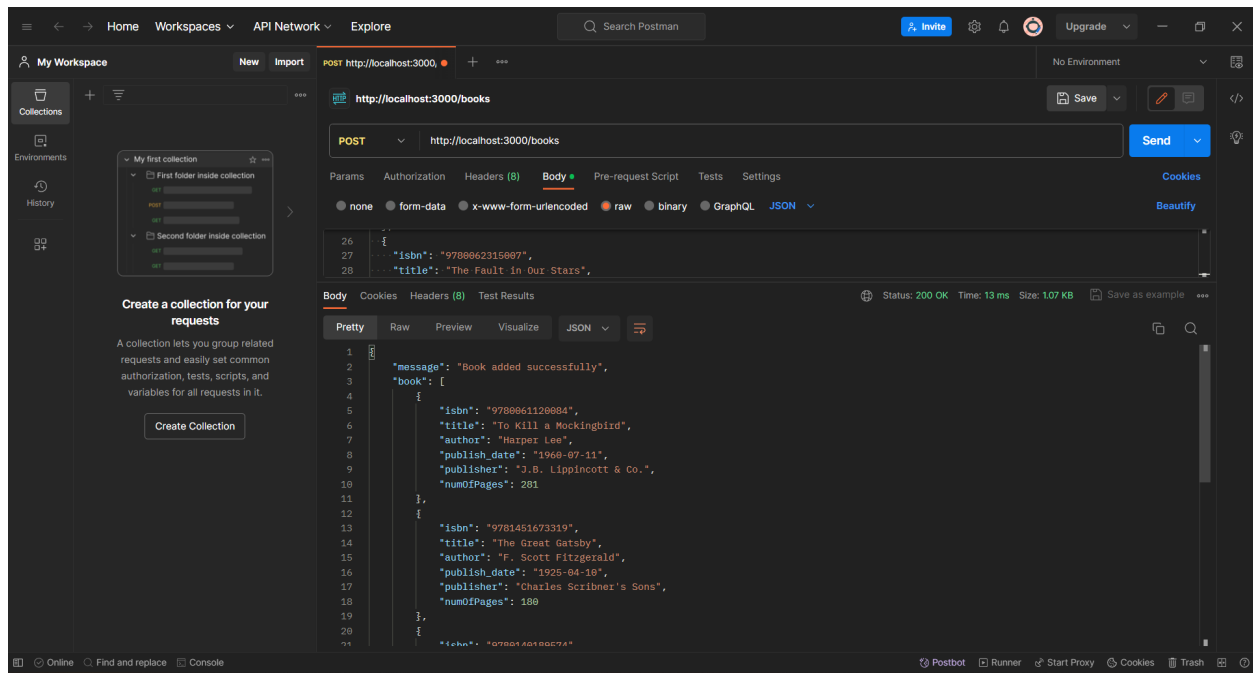
```
1 const express = require("express");
2 const bodyParser = require("body-parser");
3 const cors = require("cors");
4
5 const app = express();
6 const port = 3000;
7
8 // Middleware
9 app.use(cors());
10 app.use(bodyParser.urlencoded({ extended: false }));
11 app.use(bodyParser.json());
12
13 // Routes
14 const booksRouter = require("./routes/books");
15 app.use("/books", booksRouter);
16
17 app.listen(port, () => {
18   console.log(`Server is running on http://localhost:${port}`);
19 });
```

The terminal at the bottom shows the command 'node app.js' being executed, and the output 'Server is running on http://localhost:3000'.

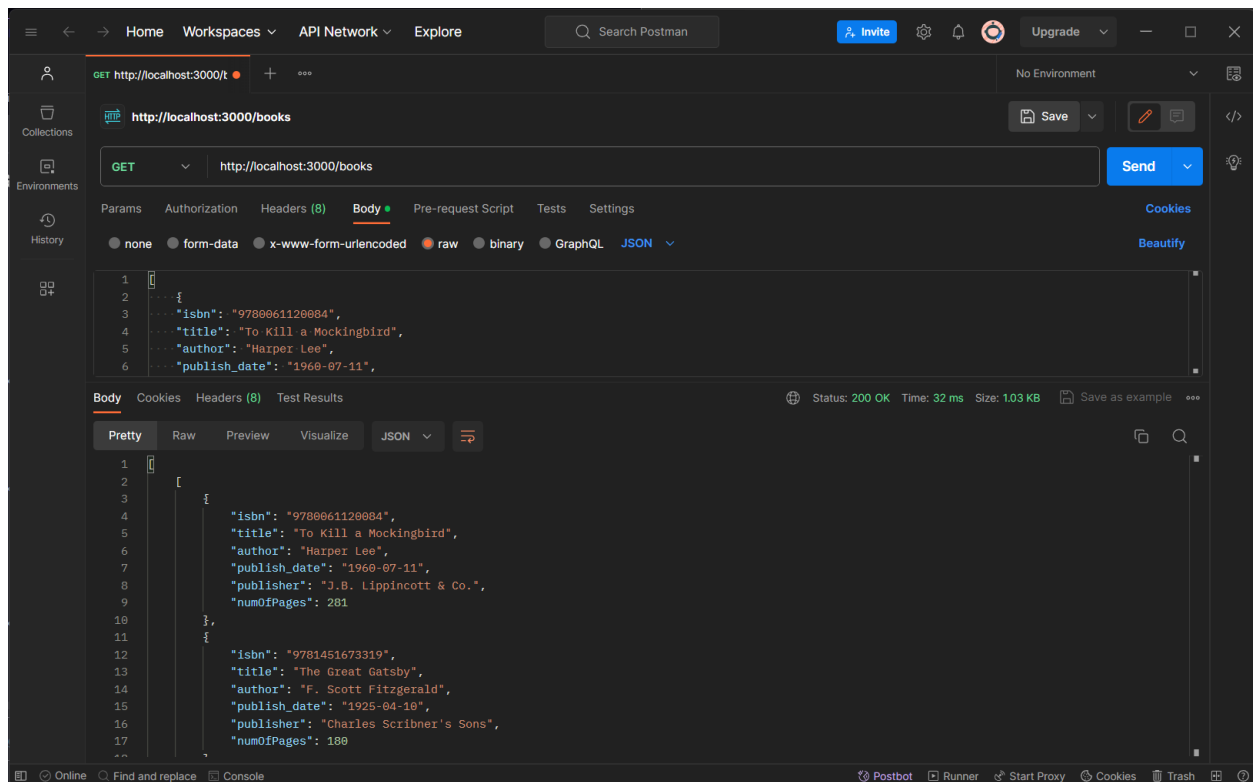
2) Send Req to Postman to add 1 book record.



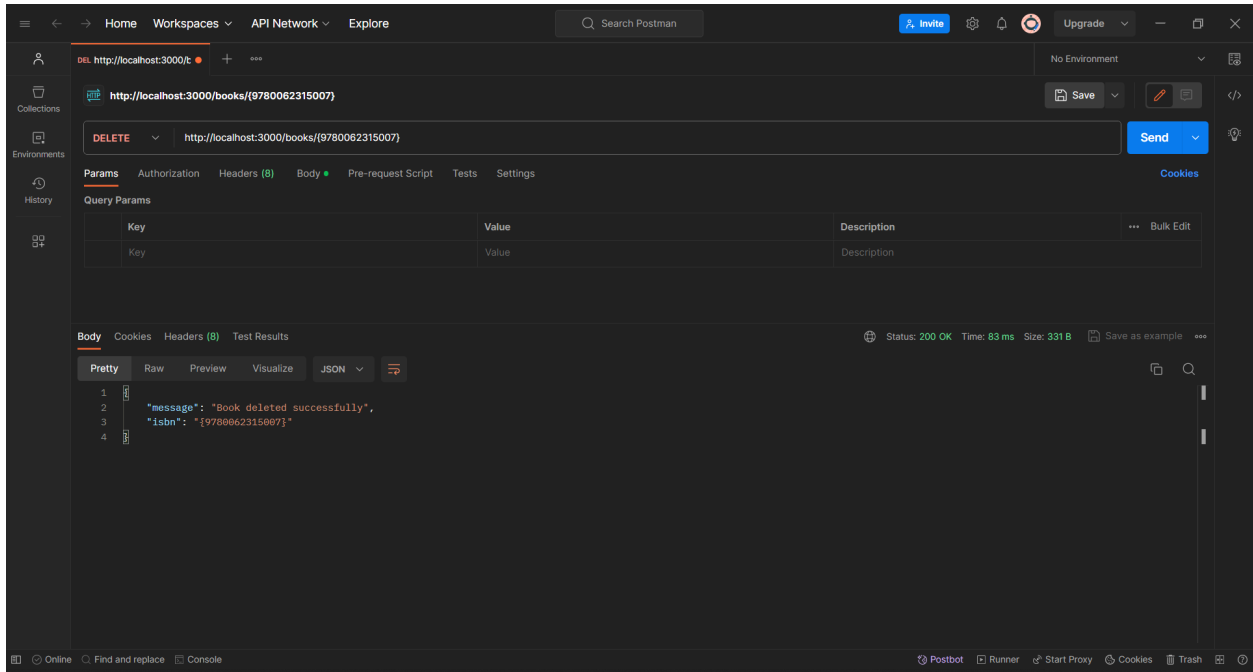
3) Add 4 books more.



4) Show all records



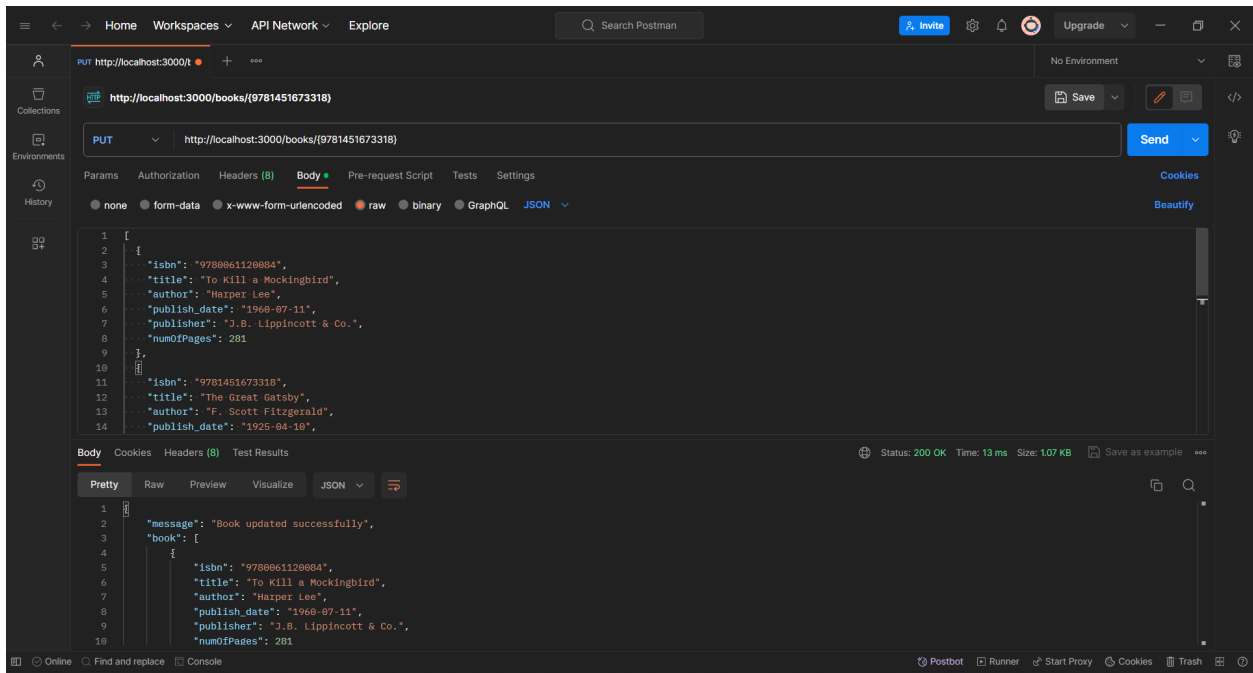
5) Book Deleted



Postman interface showing a DELETE request to `http://localhost:3000/books/9780062315007`. The response is a JSON object:

```
{
  "message": "Book deleted successfully",
  "isbn": "9780062315007"
}
```

6) Book Updated



Postman interface showing a PUT request to `http://localhost:3000/books/9781451673318`. The request body is a JSON object:

```
{
  "isbn": "9780061126684",
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "publish_date": "1960-07-11",
  "publisher": "J.B. Lippincott & Co.",
  "numOfPages": 281
}
```

The response is a JSON object:

```
{
  "message": "Book updated successfully",
  "book": {
    "isbn": "9780061126684",
    "title": "To Kill a Mockingbird",
    "author": "Harper Lee",
    "publish_date": "1960-07-11",
    "publisher": "J.B. Lippincott & Co.",
    "numOfPages": 281
  }
}
```

GIT: <https://github.com/yashtekavade/fsd/tree/main/6/server>