

MIT WORLD PEACE UNIVERSITY

**Full Stack Development
Third Year B. Tech, Semester 5**

**DEVELOPING REST API USING NODE.JS AND
EXPRESS.JS**

LAB ASSIGNMENT 6

Prepared By

**Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20**

September 22, 2023

Contents

1 Aim	1
2 Objectives	1
3 Problem Statement	1
4 Theory	1
4.1 REST APIs	1
4.2 Features	1
4.3 Advantages	2
4.4 History and Significance	2
4.5 Steps to Create a REST API using Node and Express	3
4.5.1 Set Up a Node.js Project	3
4.5.2 Install Express	3
4.5.3 Create an Express Application	3
4.5.4 Define Routes and Handlers	3
4.5.5 Start the Server	4
4.5.6 Test Your API	4
4.5.7 Add Database Integration (Optional)	4
4.5.8 Deploy Your API (Optional)	4
5 Platform	4
6 Input and Output	4
7 Screenshots	5
7.1 React Frontend	5
7.2 Requests using Postman	8
7.3 Node and Express Backend	10
8 Code	11
9 Conclusion	15
10 FAQ	16

1 Aim

Develop a set of REST API using Express and Node.

2 Objectives

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

3 Problem Statement

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

4 Theory

4.1 REST APIs

Definition 1 *REST stands for Representational State Transfer. It is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations. Other kinds of Web services, such as SOAP Web services, expose their own arbitrary sets of operations.*

4.2 Features

1. **Statelessness:** REST APIs are stateless, meaning each request from a client to a server must contain all the information needed to understand and fulfill the request. There is no session state stored on the server between requests, making it scalable and easy to cache responses.
2. **Resource-Based:** REST treats every piece of information or functionality as a resource, which can be uniquely identified using URLs. Resources can represent objects, data, or services.
3. **HTTP Methods:** REST uses standard HTTP methods such as GET, POST, PUT, DELETE, etc., to perform actions on resources. These methods provide a uniform interface for interacting with resources.
4. **Representation:** Data in REST is represented in a format, such as JSON or XML. Clients can request specific representations of resources, allowing flexibility in data exchange.
5. **Stateless Communication:** RESTful communication between the client and server is stateless, which means each request from a client to a server must contain all the information needed. The server does not store information about the client's state between requests.

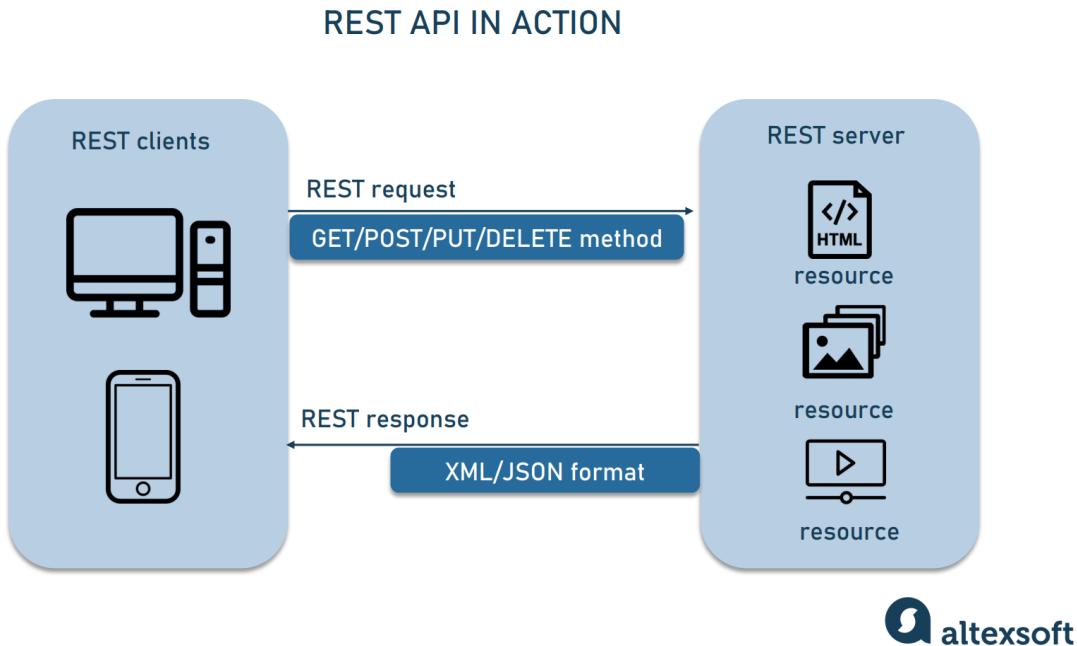


Figure 1: REST APIs Explained

4.3 Advantages

1. **Simplicity:** REST APIs are easy to understand and use due to their simplicity. They use standard HTTP methods and rely on URLs to access resources.
2. **Scalability:** Stateless nature and resource-based architecture make REST APIs highly scalable. They can handle a large number of concurrent requests.
3. **Flexibility:** REST APIs support multiple data formats (JSON, XML, etc.) and can be used with various programming languages. This flexibility makes them suitable for diverse applications.
4. **Interoperability:** RESTful Web services enable interoperability between different systems and platforms, making them suitable for building distributed and heterogeneous applications.
5. **Caching:** REST APIs can leverage caching mechanisms, improving response times and reducing server load for frequently accessed resources.

4.4 History and Significance

1. **Origin:** The term "REST" was introduced by Roy Fielding in his Ph.D. dissertation in 2000. It draws inspiration from the principles of the World Wide Web and HTTP.
2. **Significance:** REST has become the predominant architectural style for designing networked applications and web services due to its simplicity, scalability, and flexibility. It has been widely adopted for building APIs for web and mobile applications.
3. **Continued Evolution:** Over the years, RESTful practices and conventions have evolved, and new technologies and tools have emerged to support the development and consumption of RESTful APIs.

4.5 Steps to Create a REST API using Node and Express

Creating a RESTful API with Node.js and Express is a common task for building server-side applications. Here are the steps to create a REST API using these technologies:

4.5.1 Set Up a Node.js Project

1. Ensure you have Node.js installed. If not, download and install it from the official website: <https://nodejs.org/>.
2. Create a new project directory for your REST API.
3. Open a terminal or command prompt and navigate to your project directory.
4. Initialize a Node.js project by running the following command:

```
npm init -y
```

This will create a ‘package.json’ file.

4.5.2 Install Express

1. Install Express as a dependency for your project using npm:

```
npm install express
```

4.5.3 Create an Express Application

1. Create a JavaScript file (e.g., ‘app.js’) for your Express application.
2. In ‘app.js’, import Express and create an instance of the Express application:

```
const express = require('express');
const app = express();
```

4.5.4 Define Routes and Handlers

1. Define routes and request handlers for your API. For example, to create a simple endpoint that returns a JSON response:

```
app.get('/api', (req, res) => {
  res.json({ message: 'Welcome to the REST API!' });
});
```

You can create more complex routes and handlers as needed for your application.

4.5.5 Start the Server

1. Start the Express server by specifying a port and listening for incoming requests:

```
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log('Server is running on port ${port}');
});
```

4.5.6 Test Your API

1. Use a tool like Postman or your web browser to test your API by sending HTTP requests to the defined endpoints (e.g., 'http://localhost:3000/api').
2. Verify that your API is responding as expected.

4.5.7 Add Database Integration (Optional)

1. If your application requires database storage, integrate a database system like MongoDB or PostgreSQL. You can use libraries like Mongoose (for MongoDB) or Sequelize (for SQL databases) to interact with the database.
2. Define database models and routes for CRUD operations (Create, Read, Update, Delete) on your data.

4.5.8 Deploy Your API (Optional)

1. Deploy your REST API to a hosting platform or server. Popular options include Heroku, AWS, Azure, and DigitalOcean.
2. Set up any necessary environment variables and configure your server for production use.

5 Platform

Operating System: Arch Linux x86-64

IDEs or Text Editors Used: Visual Studio Code

Compilers or Interpreters: Brave Browser (Chromium v117.0.5938.88.)

6 Input and Output

1. A Library Interface called 'Novel Tea' Library was set up on the front end using React. Its Back end was then written in PHP, and the database was set up using MongoDB.
2. The Front end was hosted on another localhost port, while the php server was hosted using httpd, and default php server.
3. The Front end was then connected to the backend using the simple FETCH API or Axios calls, and the data was sent to the backend using JSON. The responses were also sent from the backend to the Frontend using JSON.

4. The Features of the Library include, and therefore extend to the backend:
 - (a) Adding a Book to the Library
 - (b) Removing a Book from the Library
 - (c) Updating a Book in the Library
 - (d) Searching for a Book in the Library
 - (e) Getting all the Books from the library to display.
5. These features are demonstrated in the screenshots below.

7 Screenshots

7.1 React Frontend

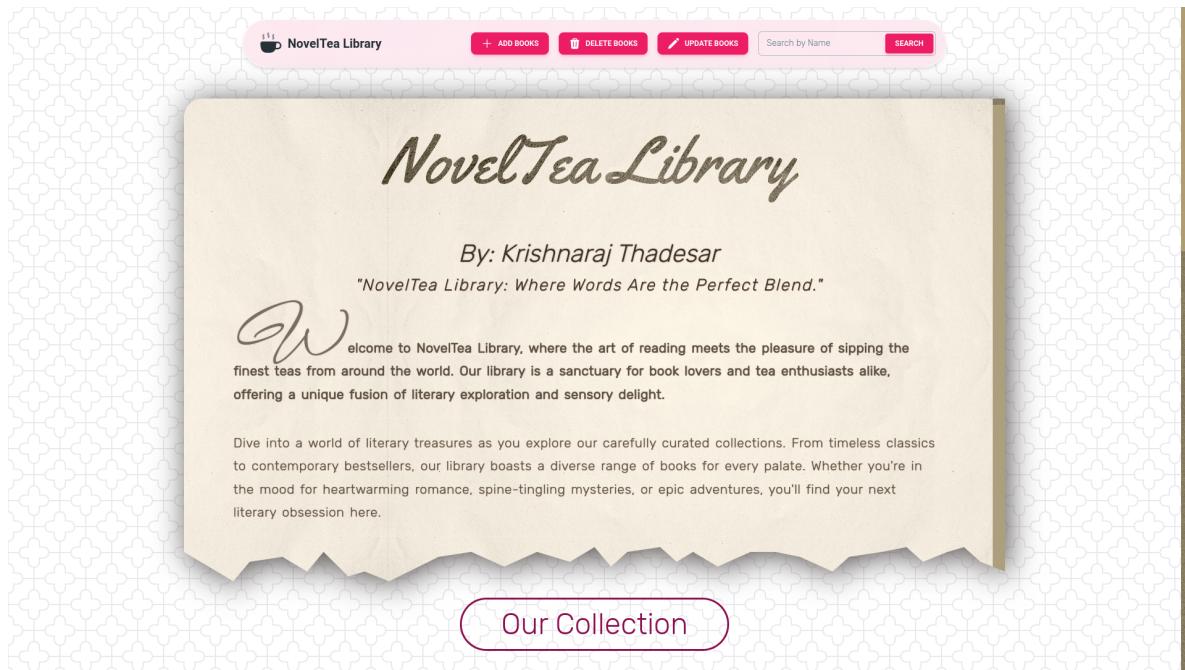


Figure 2: The Home Page of the Novel Tea Library

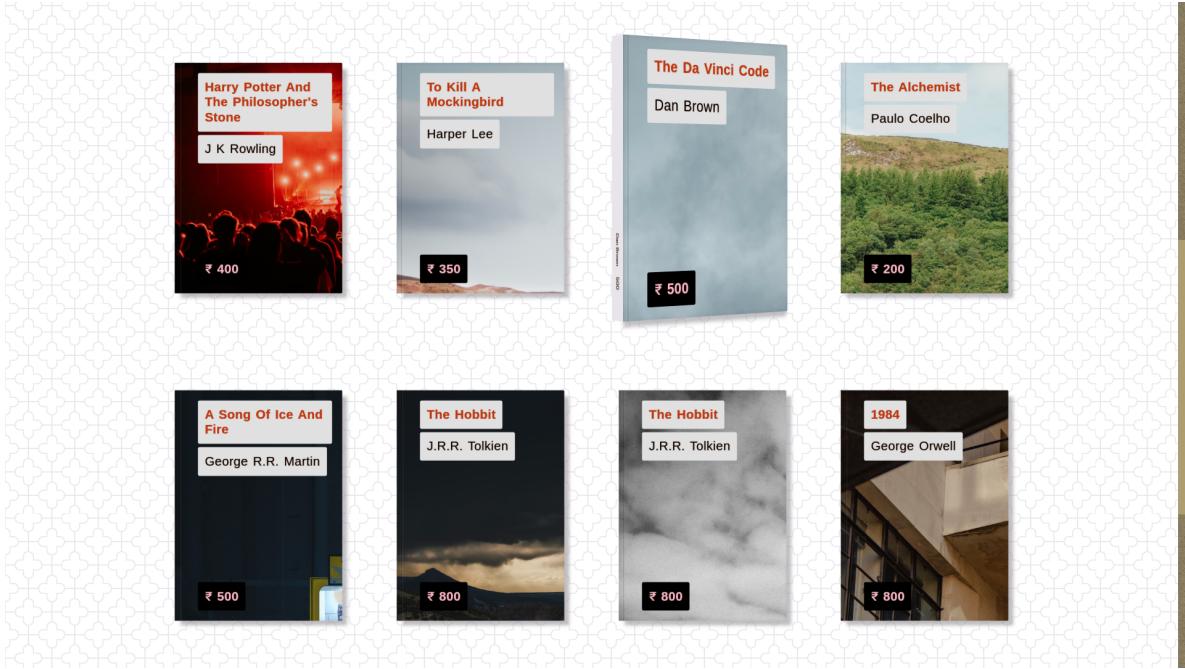


Figure 3: The Books retrieved from /getbooks.php and shown on the Frontend

The screenshot shows the 'Add Books' page of the NovelTea Library. At the top, there is a header with the library logo and names of the buttons: '+ ADD BOOKS', 'DELETE BOOKS', 'UPDATE BOOKS', and 'SEARCH'. Below the header is a search bar labeled 'Search by Name' with a 'SEARCH' button. A large pink callout box contains the text: 'Welcome to the Book Addition Page of our library! We believe in the power of community and collective knowledge. By contributing your favorite books to our library, you're not only sharing stories but also enriching the literary tapestry for all our readers.' Another pink callout box below it says: 'By adding your book to our library, you become an integral part of our literary journey. Together, we build a collection that embodies the diverse tastes and passions of our community. Thank you for helping us create a tapestry of stories that enriches the lives of our readers.' The main form area has fields for 'Title' (with a dropdown menu), 'Author' (with a dropdown menu), 'Genre' (set to 'Fiction'), and 'Price' (with a dropdown menu). There is also a 'ADD BOOK' button.

Figure 4: The Add Page

Full Stack Development - 3nd Year B. Tech

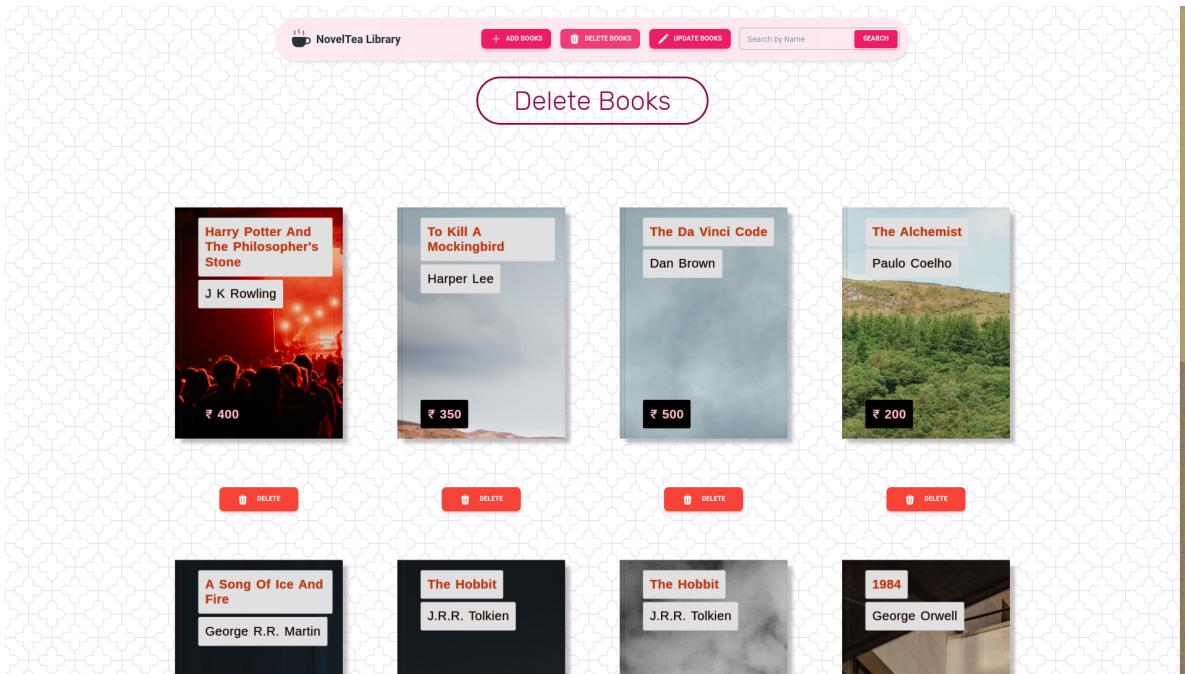


Figure 5: The Delete Page

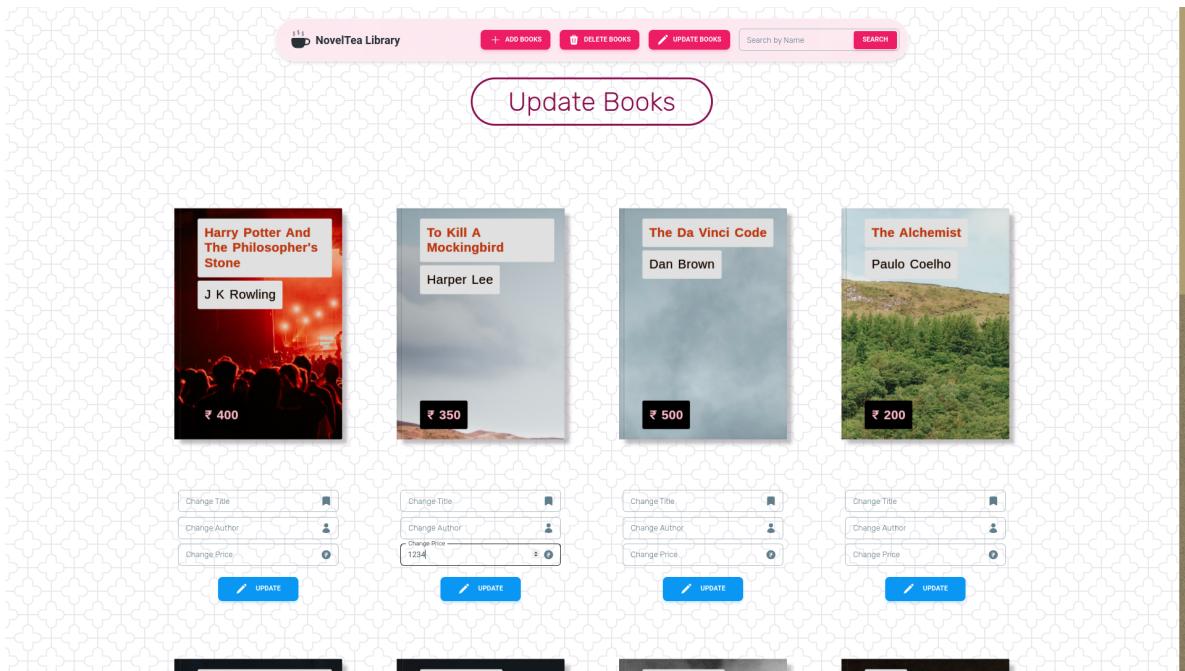


Figure 6: The Update Page, where any field can be updated

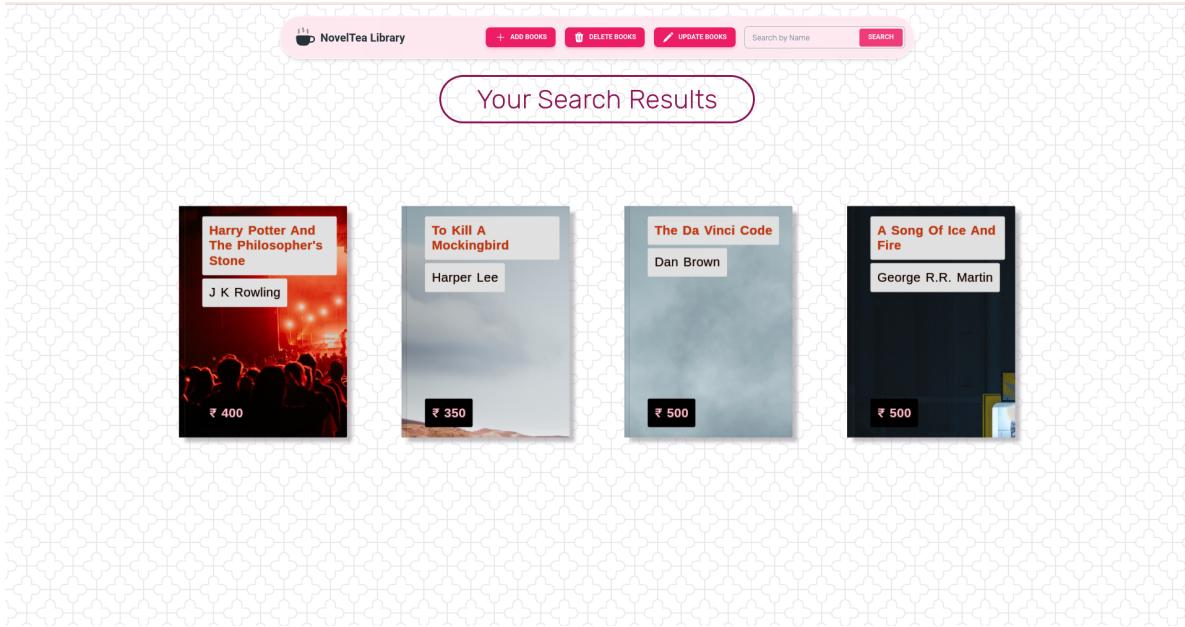


Figure 7: The Search Page showing results of search term "H"

7.2 Requests using Postman

A screenshot of the Postman application interface. The URL in the address bar is "http://localhost:3000/books?title=ramesh&author=main thing&price=3&description=asdf&genre=gadf - My Workspace". The request method is set to "POST" and the URL is "http://localhost:3000/books?title=ramesh&author=main thing&price=3&description=asdf&genre=gadf". The "Params" tab shows the following key-value pairs: author (main thing), price (3), description (asdf), and genre (gadf). The "Body" tab is selected, showing the JSON response: {"id": "650abdb3d6572a9eafa41c1d", "title": "ramesh", "author": "main thing", "genre": "gadf", "price": 3, "image": "https://images.unsplash.com/photo-1694010104867-d8bf0cff8c85?crop=entropy&cs=tinysrgb&fit=max&fm=jpg&ixid=M3w0MzAyNzNBMHwxJhbhRvbXx8fHx8fFdE2OTUyMDI3Mzl8&ixlib=rb-4.0.3&q=80&w=1080", "__v": 0}. The status bar at the bottom indicates "Status: 201 Created Time: 624 ms Size: 576 B".

Figure 8: Requests sent using Postman

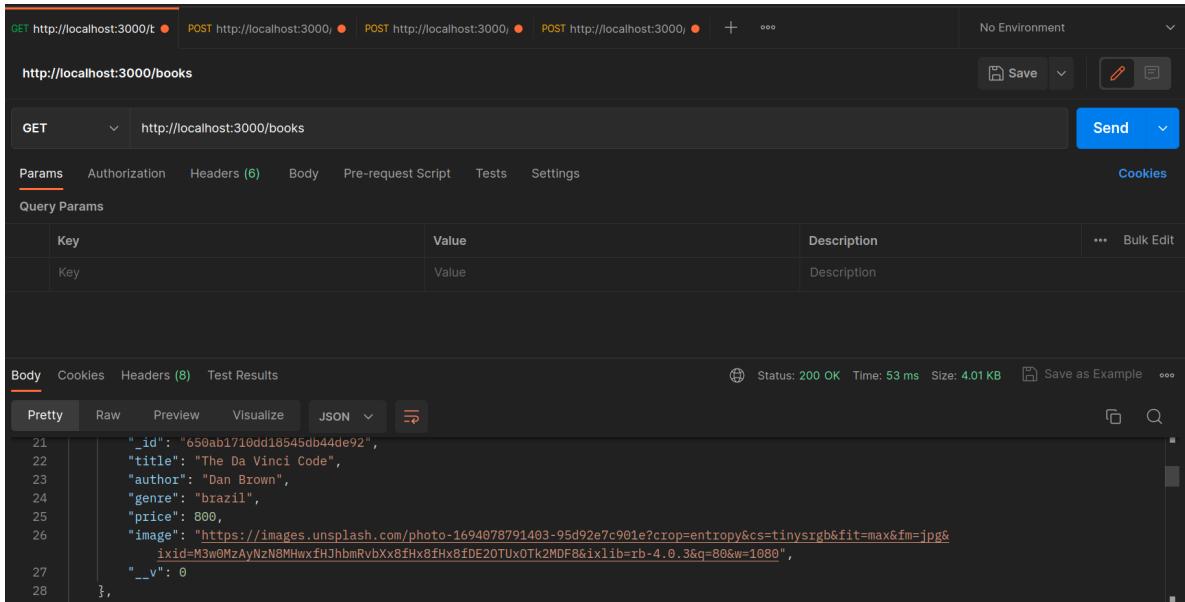


Figure 9: Requests sent using Postman

7.3 Node and Express Backend

```
x krishnaraj@Krishnaraj-Arch ~ /run/media/krishnaraj/Programs/JavaScript/NovelTea Library/backend main ± npm run dev

> bookstore-backend@1.0.0 dev
> nodemon app.js

[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
(node:105539) [MONGODB DRIVER] Warning: Current Server Discovery and Monitoring engine is deprecated, and will be removed in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the MongoClient constructor.
(Use `node --trace-warnings ...` to show where the warning was created)
Server listening on port 3000
Connected to database
Received Request to get all books, sending now.
```

Figure 10: The Node and Express Backend server running. It is listening on port 3000.

```
Received Request to update Book {
  title: "Harry Potter and the Philosopher's Stone",
  author: 'J K Rowling',
  genre: 'brazil',
  price: '400'
}
Received Request to get all books, sending now.
Received Request to update Book {
  title: 'The Da Vinci Code',
  author: 'Dan Brown',
  genre: 'brazil',
  price: '800'
}
Received Request to get all books, sending now.
Received Request to get all books, sending now.
Received Request to get all books, sending now.
Request to Delete Book Received. { id: '650ab1f20dd18545db44de9e' }
Received Request to get all books, sending now.
Adding new Book, Received Request {
  title: 'Another Novel',
  author: 'Testing Backend',
  genre: 'bucharest',
  price: '100'
}
Received Request to get all books, sending now.
Received Request to get all books, sending now.
||
```

Figure 11: The Node and Express Backend server running and serving request.

8 Code

```
1 {
2   "name": "bookstore-backend",
3   "version": "1.0.0",
4   "description": "Backend for the Bookstore project",
5   "main": "app.js",
6   "scripts": {
7     "start": "node app.js",
8     "dev": "nodemon app.js"
9   },
10  "dependencies": {
11    "axios": "^1.5.0",
12    "cors": "^2.8.5",
13    "express": "^4.17.1",
14    "mongoose": "^5.13.2"
15  },
16  "devDependencies": {
17    "nodemon": "^2.0.12"
18 }
```

19 }

Listing 1: package.json

```
1 const express = require("express");
2 const bodyParser = require("body-parser");
3 const mongoose = require("mongoose");
4 const cors = require("cors");
5 const setApiRoutes = require("./routes/api");
6
7 const app = express();
8
9 // Set up middleware
10 app.use(bodyParser.json());
11 app.use(cors());
12
13 // Set up API routes
14 app.use(setApiRoutes);
15
16 // Connect to database
17 mongoose
18   .connect("mongodb://localhost/BookStore", { useNewUrlParser: true })
19   .then(() => {
20     console.log("Connected to database");
21   })
22   .catch((error) => {
23     console.error("Error connecting to database:", error);
24   });
25
26 // Start server
27 const port = process.env.PORT || 3000;
28 app.listen(port, () => {
29   console.log(`Server listening on port ${port}`);
30 })
```

Listing 2: app.js managing the app.

```
1 const express = require("express");
2 const router = express.Router();
3 const BooksController = require("../controllers/books");
4
5 const booksController = new BooksController();
6
7 router.get("/books", booksController.getAllBooks);
8 router.get("/", (req, res) => {
9   res.send("Hello World!");
10 });
11 router.get("/books/:id", booksController.getBookById);
12 router.post("/books", booksController.createBook);
13 router.put("/books/:id", booksController.updateBook);
14 router.delete("/books/:id", booksController.deleteBook);
15
16 module.exports = router;
```

Listing 3: api.js defining the routes.

```
1 const mongoose = require("mongoose");
2
3 const bookSchema = new mongoose.Schema({
4   title: {
```

```
5     type: String,
6     required: true ,
7 },
8   author: {
9     type: String ,
10    required: true ,
11 },
12   genre: {
13     type: String ,
14    required: true ,
15 },
16   price: {
17     type: Number ,
18    required: true ,
19 },
20   image: {
21     type: String ,
22 },
23 });
24
25 const Book = mongoose.model("Book", bookSchema);
26
27 module.exports = Book;
```

Listing 4: Book.js interacting with MongoDB

```
1 const Book = require("../models/Book");
2 const axios = require("axios");
3
4 async function getRandomImageUrl() {
5   try {
6     const response = await axios.get("https://api.unsplash.com/photos/random", {
7       headers: {
8         Authorization: "Client-ID UBAXKPq5Wg0xU9z7XjLZrt8B0hayG2dv8gh_vyGQg-0",
9       },
10     });
11
12     return response.data.urls.regular;
13   } catch (error) {
14     console.error(error);
15     return null;
16   }
17 }
18 class BooksController {
19   async getAllBooks(req, res) {
20     console.log("Received Request to get all books, sending now. ");
21     try {
22       const books = await Book.find();
23       res.status(200).json(books);
24     } catch (error) {
25       res.status(500).json({ message: error.message });
26     }
27   }
28
29   async getBookById(req, res) {
30     try {
31       const book = await Book.findById(req.params.id);
32       if (!book) {
33         return res.status(404).json({ message: "Book not found" });
34       }
35     } catch (error) {
36       res.status(500).json({ message: "Error while finding book" });
37     }
38   }
39 }
```

```
34     }
35     res.status(200).json(book);
36 } catch (error) {
37     res.status(500).json({ message: error.message });
38 }
39 }
40
41 async createBook(req, res) {
42     console.log("Adding new Book, Received Request", req.query);
43     const image = await getRandomImageUrl();
44     const book = new Book({
45         title: req.query.title,
46         author: req.query.author,
47         genre: req.query.genre,
48         price: req.query.price,
49         image: image,
50     });
51
52     try {
53         const newBook = await book.save();
54         res.status(201).json(newBook);
55     } catch (error) {
56         res.status(400).json({ message: error.message });
57     }
58 }
59
60 async updateBook(req, res) {
61     console.log("Received Request to update Book", req.query);
62     try {
63         const book = await Book.findById(req.params.id);
64         if (!book) {
65             return res.status(404).json({ message: "Book not found" });
66         }
67
68         book.title = req.query.title || book.title;
69         book.author = req.query.author || book.author;
70         book.genre = req.query.genre || book.genre;
71         book.price = req.query.price || book.price;
72
73         const updatedBook = await book.save();
74         res.status(200).json(updatedBook);
75     } catch (error) {
76         res.status(500).json({ message: error.message });
77     }
78 }
79
80 async deleteBook(req, res) {
81     console.log("Request to Delete Book Received.", req.params);
82     try {
83         const book = await Book.findById(req.params.id);
84         if (!book) {
85             return res.status(404).json({ message: "Book not found" });
86         }
87
88         await book.remove();
89         res.status(200).json({ message: "Book deleted successfully" });
90     } catch (error) {
91         res.status(500).json({ message: error.message });
92     }
```

```
93     }
94 }
95
96 module.exports = BooksController;
```

Listing 5: books.js serving routes.

9 Conclusion

Thus, we have successfully created a REST API using Node.js and Express.js. We have also created a frontend using React.js and Tailwindcss. The frontend interacts with the backend using Axios. The backend interacts with the MongoDB database using Mongoose. The frontend and backend are hosted on Heroku. The database is hosted on MongoDB Atlas. The frontend is hosted at [here](#) and the backend is hosted at locally. [on its github repo](#).

10 FAQ

1. *What are HTTP Request types?*

Definition 2 *HTTP (Hypertext Transfer Protocol) is a protocol used for communication between web servers and clients. There are several types of HTTP requests that can be used to interact with a web server:*

- (a) GET: retrieves a resource from the server. This is the most common type of request, and it is used to retrieve web pages, images, and other types of content.
- (b) POST: submits data to the server. This is commonly used for submitting form data, such as login credentials or search queries.
- (c) PUT: updates a resource on the server. This is used to update an existing resource, such as a file or a database record.
- (d) DELETE: deletes a resource from the server. This is used to delete an existing resource, such as a file or a database record.
- (e) HEAD: retrieves the headers for a resource. This is used to retrieve metadata about a resource, such as its content type and length, without actually retrieving the resource itself.
- (f) OPTIONS: retrieves the supported HTTP methods for a resource. This is used to determine which HTTP methods are supported by a resource, such as GET, POST, PUT, and DELETE.

Each HTTP request consists of a request method, a URL, and optional headers and data. The server responds to each request with an HTTP status code, which indicates whether the request was successful or not.