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

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.

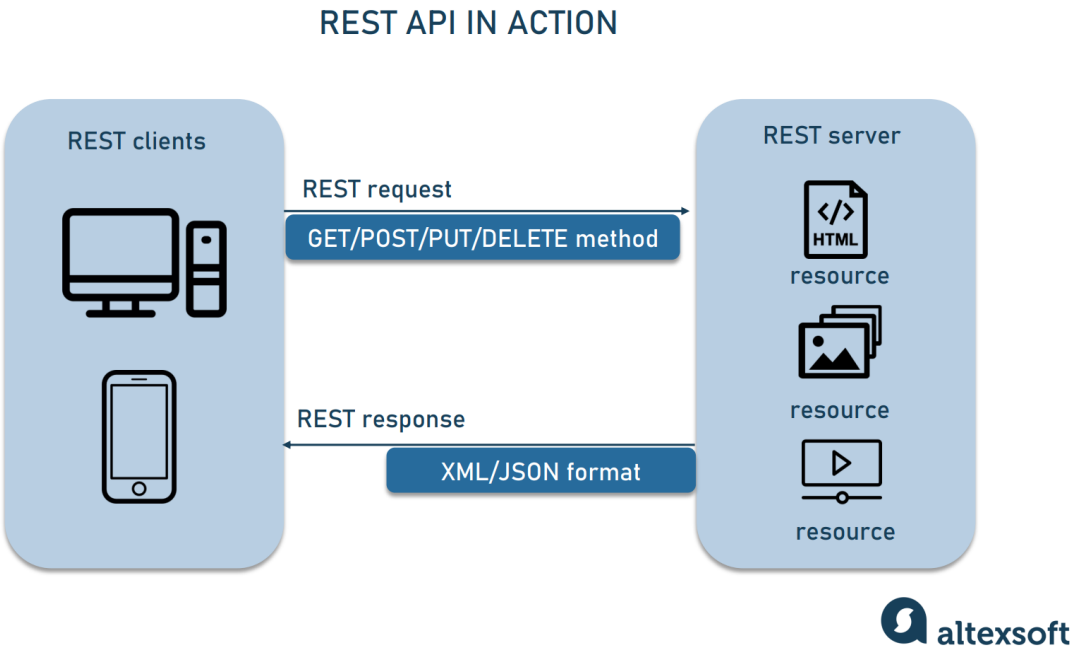

Figure 1: REST APIs Explained

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 Webpage was created for searching for images. The user can enter a search term and the number of images to be displayed. The images are fetched from the Unsplash, Pexels and Pixabay APIs, and merged into a single array. The images are then displayed on the webpage.
2. This was done so as to resolve the problem of having to browser multiple webpages to find high quality images for making presentations. This webpage allows the user to search for images from multiple sources at once.
3. The webpage was created using React and Tailwindcss. The images are fetched using the Axios library. The screenshot of the webpage is shown below.

7 Screenshots

7.1 React Frontend

7.2 React Frontend

Figure 2: The Home Page of the Novel Tea Library

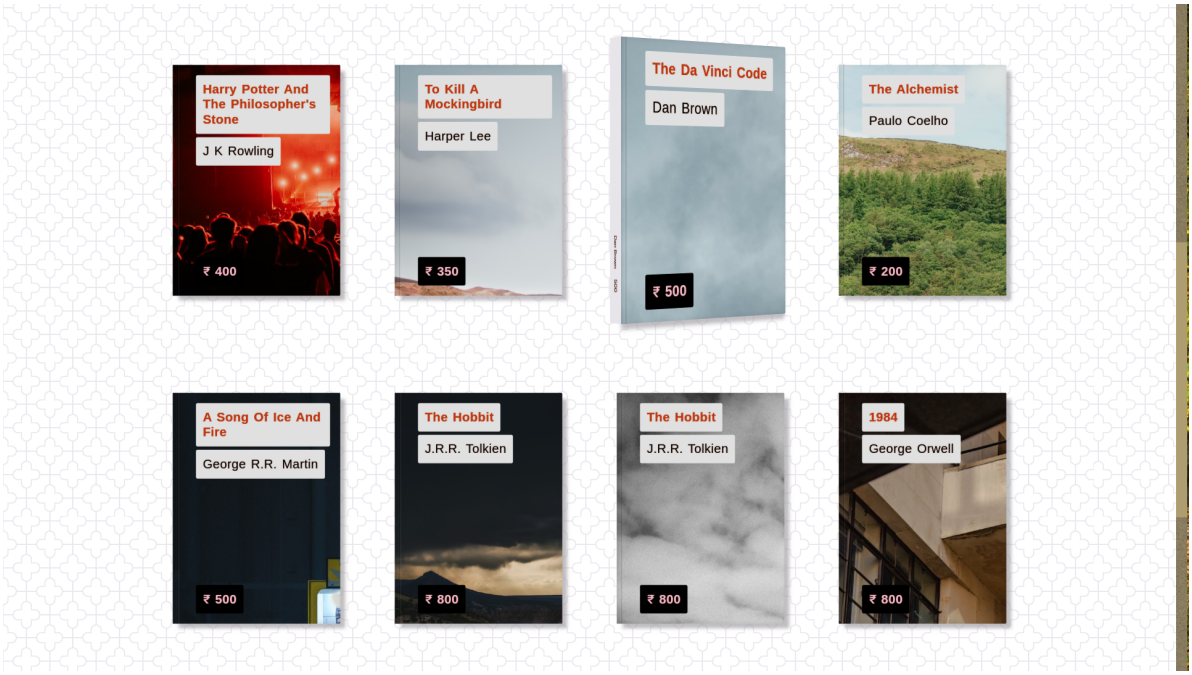

Figure 3: The Books retreived from /getbooks.php and shown on the Frontend

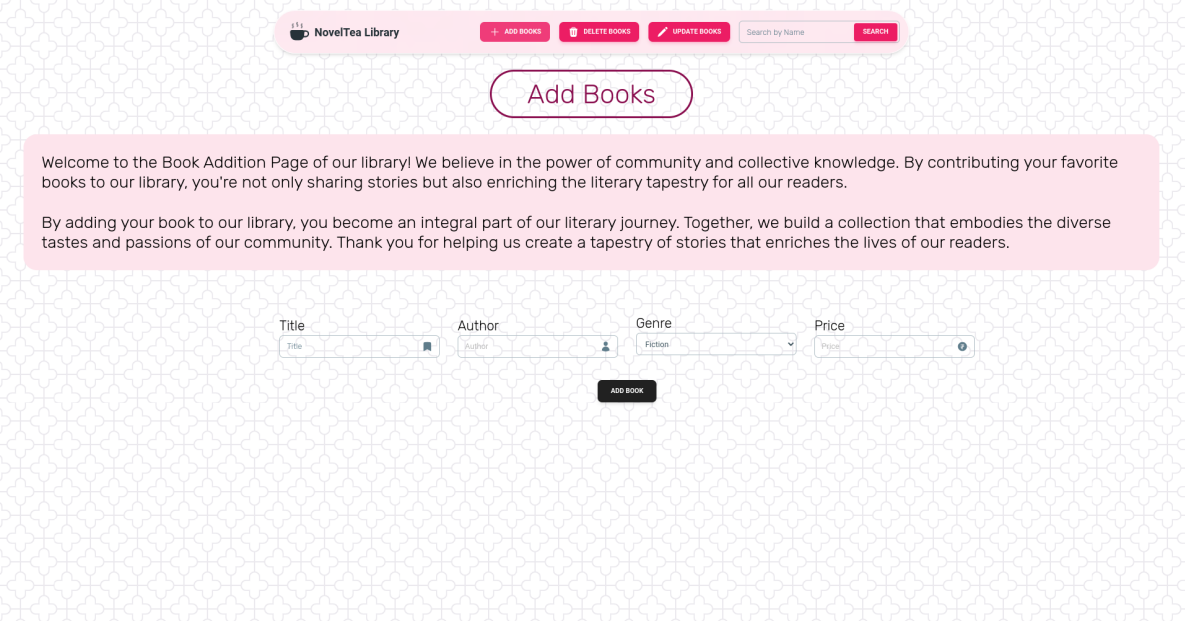

Figure 4: The Add Page

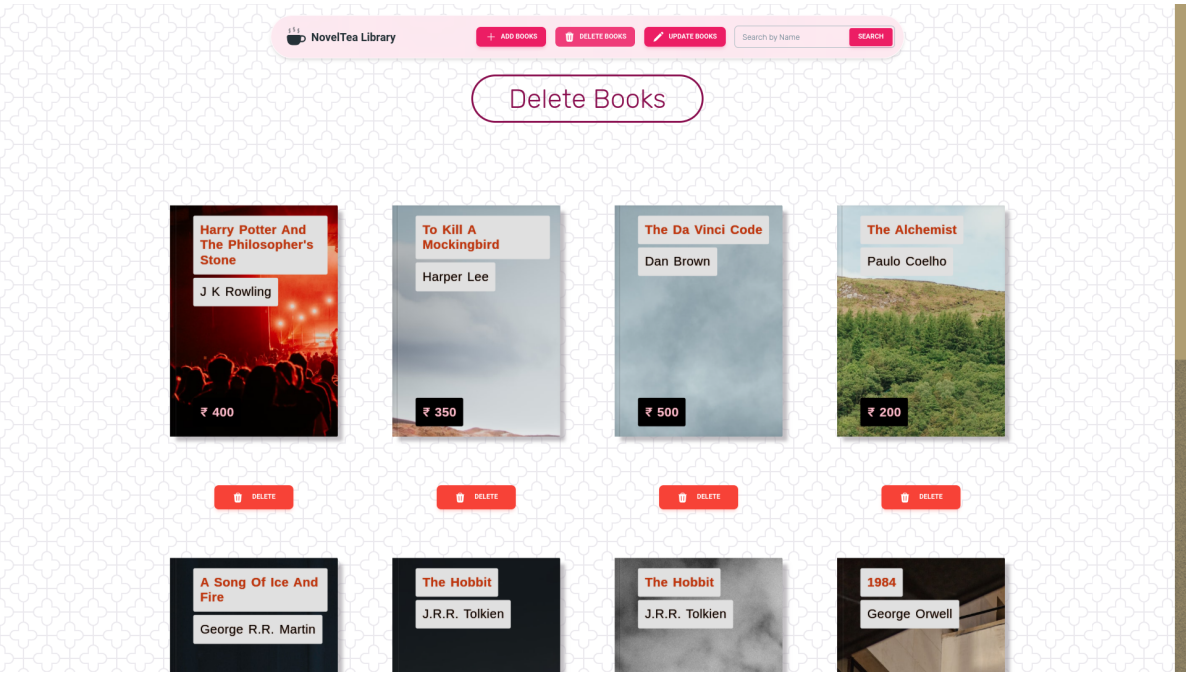

Figure 5: The Delete Page

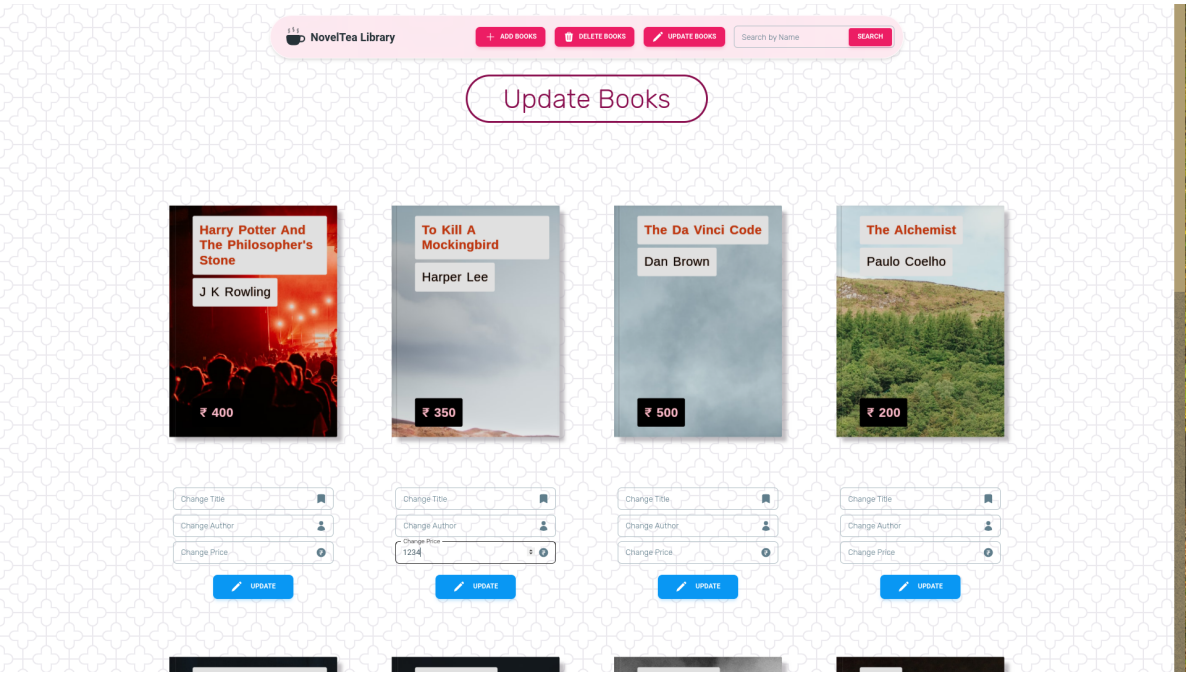

Figure 6: The Update Page, where any field can be updated

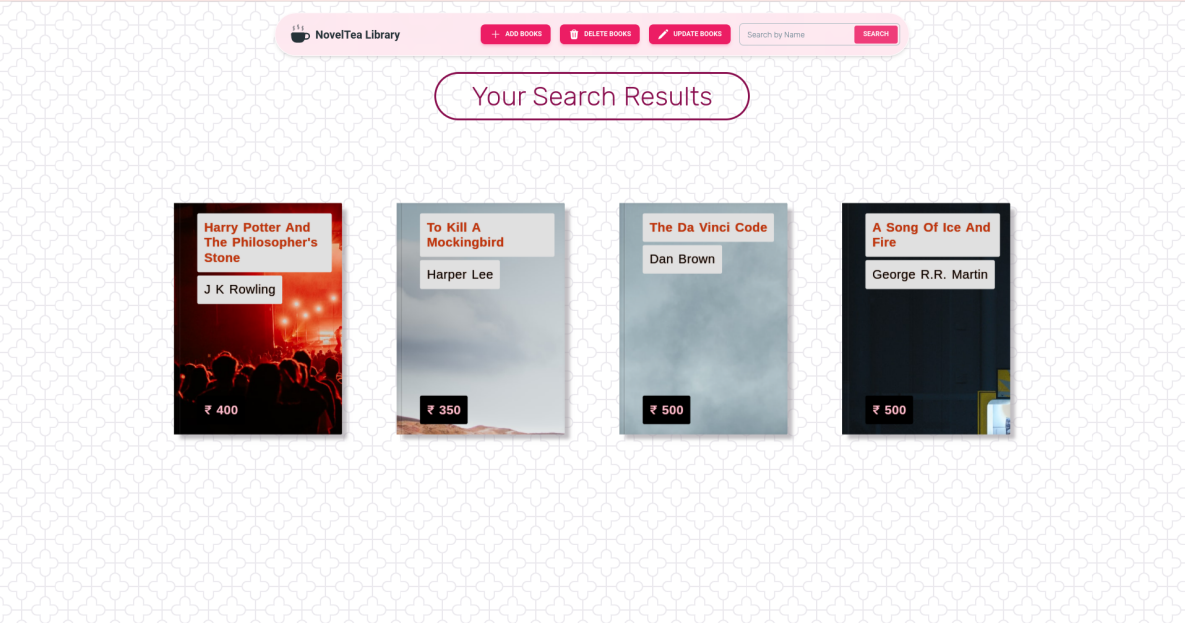

Figure 7: The Search Page showing results of search term "H"

7.3 Requests using Postman

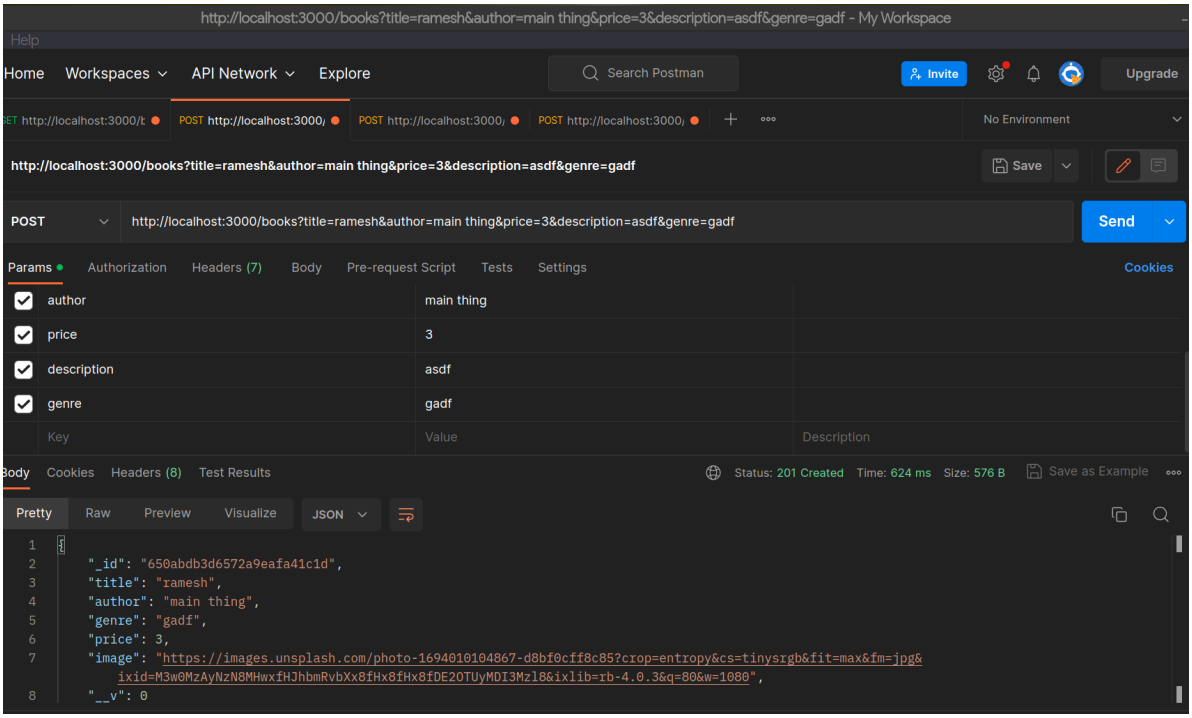

Figure 8: Requests sent using Postman

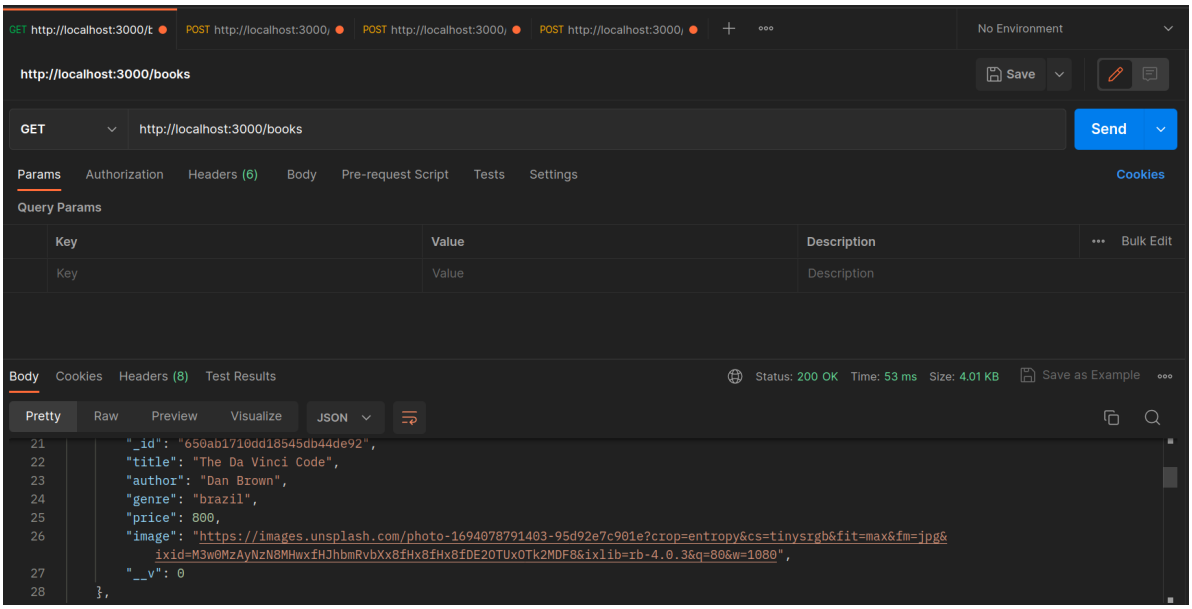

Figure 9: Requests sent using Postman

7.4 Node and Express Backend

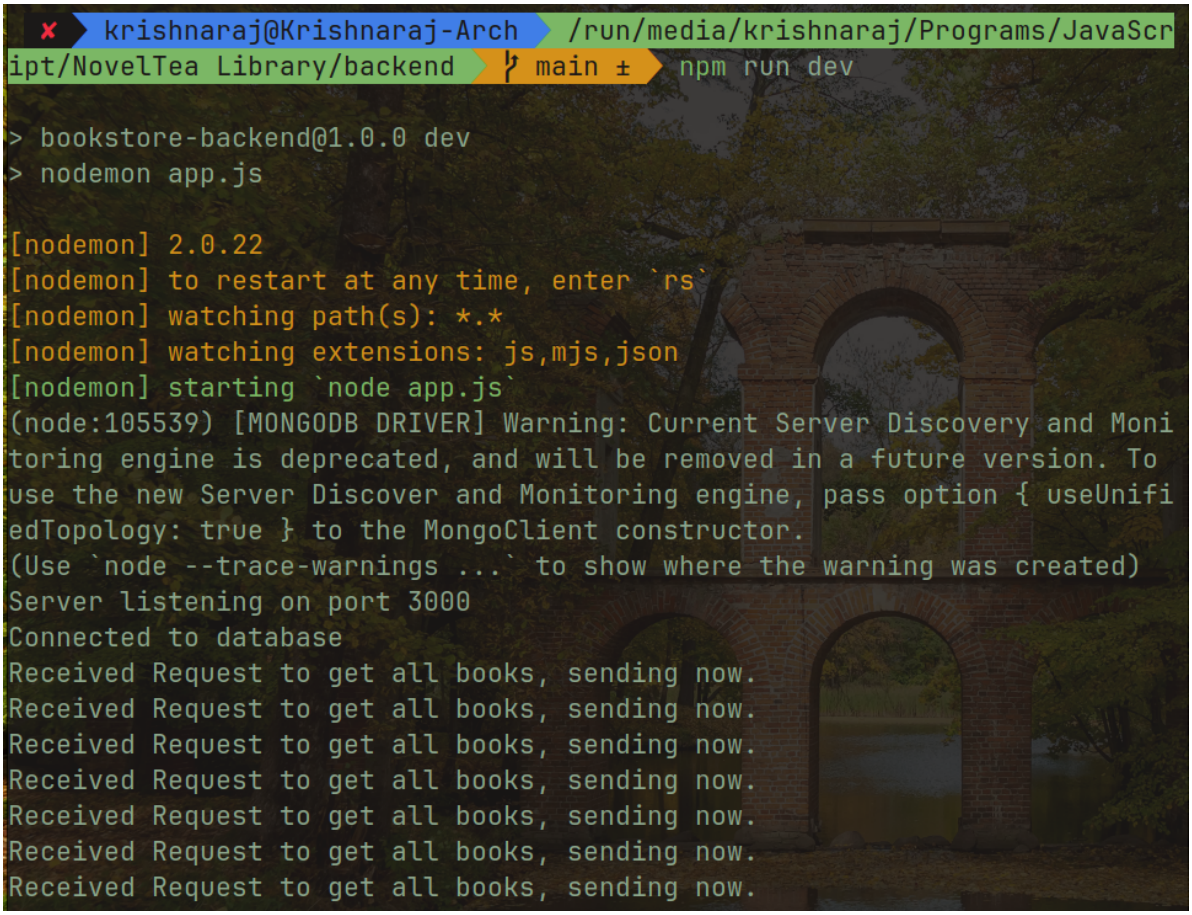

Figure 10: The Node and Express Backend server running. It is listening on port 3000.

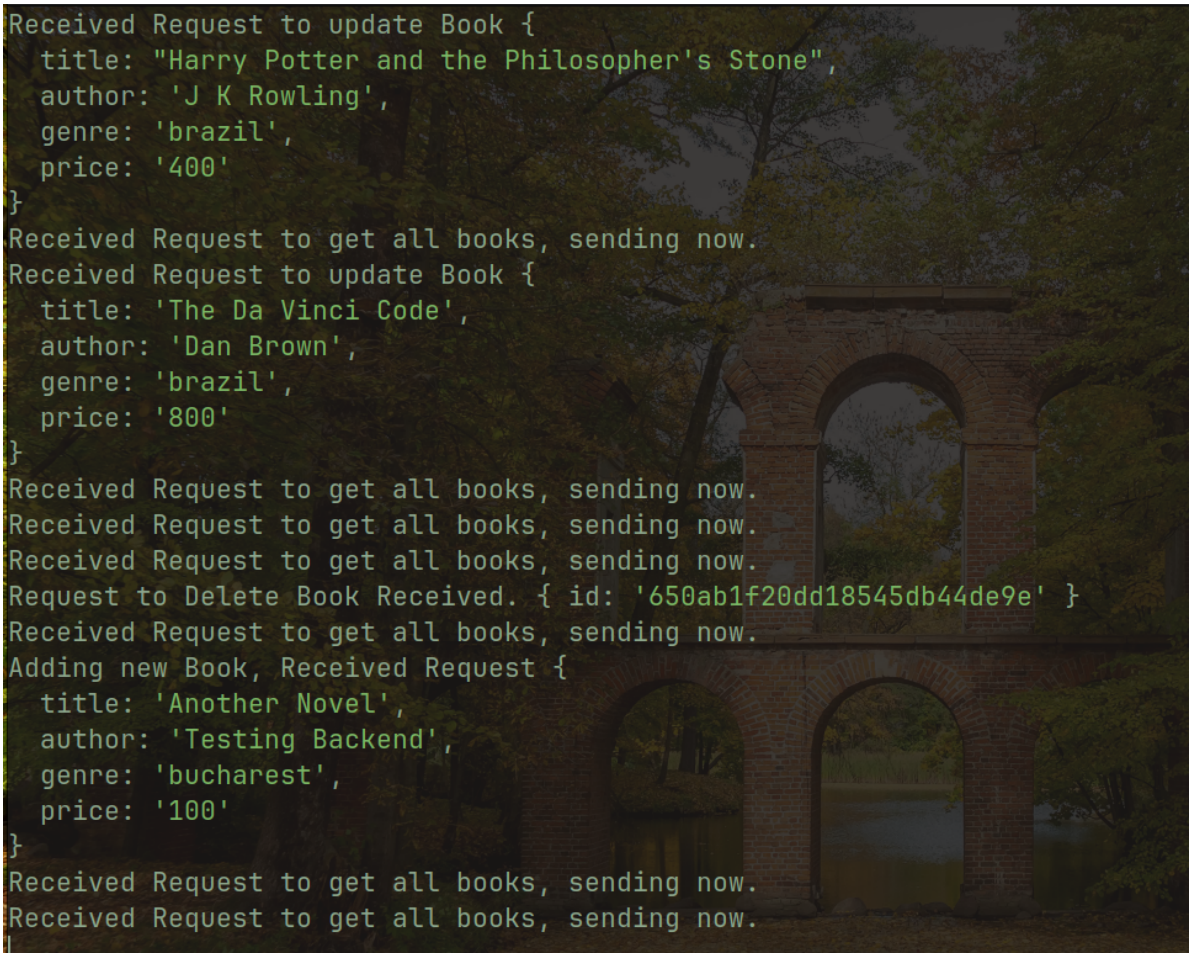

Figure 11: The Node and Express Backend server running and serving request.

8 Code

```
{
  "name": "bookstore-backend",
  "version": "1.0.0",
  "description": "Backend for the Bookstore project",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js"
  },
  "dependencies": {
    "axios": "^1.5.0",
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "mongoose": "^5.13.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.12"
  }
```

```
}
```

Listing 1: package.json

```
const express = require("express");
const bodyParser = require("body-parser");
const mongoose = require("mongoose");
const cors = require("cors");
const setApiRoutes = require("./routes/api");

const app = express();

// Set up middleware
app.use(bodyParser.json());
app.use(cors());

// Set up API routes
app.use(setApiRoutes);

// Connect to database
mongoose
  .connect("mongodb://localhost/BookStore", { useNewUrlParser: true })
  .then(() => {
    console.log("Connected to database");
  })
  .catch((error) => {
    console.error("Error connecting to database:", error);
  });

// Start server
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});
```

Listing 2: app.js managing the app.

```
const express = require("express");
const router = express.Router();
const BooksController = require("../controllers/books");

const booksController = new BooksController();

router.get("/books", booksController.getAllBooks);
router.get("/", (req, res) => {
  res.send("Hello World!");
});
router.get("/books/:id", booksController.getBookById);
router.post("/books", booksController.createBook);
router.put("/books/:id", booksController.updateBook);
router.delete("/books/:id", booksController.deleteBook);

module.exports = router;
```

Listing 3: api.js defining the routes.

```
const mongoose = require("mongoose");

const bookSchema = new mongoose.Schema({
  title: {
```

```
    type: String,
    required: true,
  },
  author: {
    type: String,
    required: true,
  },
  genre: {
    type: String,
    required: true,
  },
  price: {
    type: Number,
    required: true,
  },
  image: {
    type: String,
  },
});

const Book = mongoose.model("Book", bookSchema);

module.exports = Book;
```

Listing 4: Book.js interacting with MongoDB

```
const Book = require("../models/Book");
const axios = require("axios");

async function getRandomImageUrl() {
  try {
    const response = await axios.get("https://api.unsplash.com/photos/random", {
      headers: {
        Authorization: "Client-ID UBAXKPq5WgOxU9z7XjLZrt8B0hayG2dv8gh_vyGQg-0",
      },
    });

    return response.data.urls.regular;
  } catch (error) {
    console.error(error);
    return null;
  }
}
class BooksController {
  async getAllBooks(req, res) {
    console.log("Received Request to get all books, sending now. ");
    try {
      const books = await Book.find();
      res.status(200).json(books);
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  }

  async getBookById(req, res) {
    try {
      const book = await Book.findById(req.params.id);
      if (!book) {
        return res.status(404).json({ message: "Book not found" });
```

```
      }
      res.status(200).json(book);
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  }

  async createBook(req, res) {
    console.log("Adding new Book, Received Request", req.query);
    const image = await getRandomImageUrl();
    const book = new Book({
      title: req.query.title,
      author: req.query.author,
      genre: req.query.genre,
      price: req.query.price,
      image: image,
    });

    try {
      const newBook = await book.save();
      res.status(201).json(newBook);
    } catch (error) {
      res.status(400).json({ message: error.message });
    }
  }

  async updateBook(req, res) {
    console.log("Received Request to update Book", req.query);
    try {
      const book = await Book.findById(req.params.id);
      if (!book) {
        return res.status(404).json({ message: "Book not found" });
      }

      book.title = req.query.title || book.title;
      book.author = req.query.author || book.author;
      book.genre = req.query.genre || book.genre;
      book.price = req.query.price || book.price;

      const updatedBook = await book.save();
      res.status(200).json(updatedBook);
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  }

  async deleteBook(req, res) {
    console.log("Request to Delete Book Received.", req.params);
    try {
      const book = await Book.findById(req.params.id);
      if (!book) {
        return res.status(404).json({ message: "Book not found" });
      }

      await book.remove();
      res.status(200).json({ message: "Book deleted successfully" });
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
```

```
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 https://noveltea.surge.sh/ and the backend is hosted at https://novel-tea-library-backend.herokuapp.com/.

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.