RESTFUL API FOR AN INVENTORY MGT SYSTEM

A NAAN MUDHALVAN REPORT

SUBMITTED BY

VENKATACHALAM M (912421106020)

VIGNESHWARAN P (912421106021)

BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

SHANMUGANATHAN ENGINEERING COLLEGE

ARASAMPATTI, PUDUKKOTTAI – 622 507

YEAR & SEMESTER	:	**IV & VII**
SUBJECT CODE	:	**NM1050**
COURSE	:	**SAAS APPLIACTION**

ANNA UNIVERSITY: CHENNAI 600 025

NOV/DEC 2024

BONAFIDE CERTIFICATE

Certified that this NM report on the project **"RESTful API for an Inventory Management System"** is the Bonafide work of **"VENKATACHALAM M (912421106020) & VIGNESHWARAN P (912421106021)"** who carried out the project work under my guidance.

SIGNATURE	**SIGNATURE**
Dr. D. LATHA, ME.,	**Dr. A. MUTHU MANICKAM M.E., Ph.D**
NM COURSE COORDINATOR	**HEAD OF THE DEPARTMENT**
ASSISTANT PROFESSOR,	**ASSISTANT PROFESSOR,**
Department of Electronics & Communication Engineering,	Department of Electronics & Communication Engineering,
Shanmuganathan Engineering College,	Shanmuganathan Engineering College,
Arasampatti – 622 507	Arasampatti – 622 507

Submitted for the Poject viva-voice on ________________________

INTERNAL EXAMINER EXTERNAL EXAMINER

ACKNOWLEDGMENT

"HARD WORK NEVER FAILS" So I thank god for having gracefully blessed me to come up till now and thereby giving strength and courage to complete the poject successfully. I sincerely submit this poject report to the almighty lotus feet.

We wish to acknowledge with thanks to the significant contribution given by the management of our college chairman **"KALLVI VALLAL Mrs.Pichappa Valliammai"**, Correspondent **Dr.P.Manikandan**, and Secretary **Mr.Vishvanathan**, Shanmuganathan Engineering College, Arasampatti, for their extensive support.

I convey my indebted humble thanks to our energetic Principal **Dr. KL. Muthuramu M.E., (W.R), M.E.,(S.E), M.I.S.T.E., F.I.E., Ph.D.,** for his moral support to complete this project.

I am grateful to our Head of the Department **Dr. A. MUTHU MANICKAM M.E., Ph.D.,** and our project co-ordinator **Dr. D. LATHA, M.E.,** for thier valuable guidelines, constructive instruction, constant encouragement, unending helps that have been provided to us during the courses of the project.

I also express my heartfelt thanks to all other staff members of Electronics communication engineering Department for their support. Above all, we thank our parents, for affording us the valuable education till now.

ABSTRACT

This project aims to develop a RESTful API for an efficient, scalable inventory management system that allows seamless tracking and management of inventory items. The REST API, built to follow REST principles, provides robust support for CRUD (Create, Read, Update, Delete) operations on inventory records, empowering users to maintain an up-to-date database of items and related information. By utilizing MongoDB, a NoSQL database, this project enables flexible, high-speed data management suitable for handling extensive and diverse data formats, accommodating the scalability needs of modern applications. The API endpoints allow data access and modification through a structured and consistent approach, reducing potential errors and maintaining data integrity across various operations. On the backend, Node.js with Express serves as the API foundation, streamlining server requests, and response cycles. Each endpoint within the REST API corresponds to a specific function in inventory management, such as adding new items, updating stock levels, and deleting obsolete records. Data retrieval is optimized through MongoDB's querying capabilities, ensuring fast, reliable access to inventory data. This structure allows the API to interact with the MongoDB database seamlessly, making it adaptable to various inventory data and minimizing latency in data processing. The frontend interface, designed with HTML, CSS, and JavaScript, serves as a user-friendly layer for interacting with the API. Through intuitive forms, buttons, and display elements, users can add, edit, view, and delete items directly from the client-side application. This RESTful API solution showcases a comprehensive understanding of full-stack development principles, highlighting the importance of RESTful design, MongoDB for data management, and dynamic front-end components in creating a cohesive, effective inventory management system. The resulting application ensures high availability, reliability, and ease of use, making it suitable for businesses seeking to streamline inventory tracking and resource management in a digital environment.

TABLE OF CONTENT

LIST OF FIGURES

CHAPTER-1

INTRODUCTION

In today’s rapidly advancing digital world, inventory management is a critical process for businesses across various sectors. As companies grow, managing inventory efficiently becomes a significant challenge, often leading to errors, resource wastage, and financial loss. Traditional methods of inventory management, such as manual record-keeping or simple spreadsheets, are no longer adequate to meet the speed and accuracy demands of modern operations. This project addresses this need by creating a RESTful API-based inventory management system, combining the efficiency and structure of a REST API with the flexibility and scalability of MongoDB. The goal is to create a centralized, user-friendly, and high-performing solution that allows seamless inventory tracking and management.

A REST API (Representational State Transfer) is a software architecture style that enables efficient communication between client and server using HTTP protocols. By adhering to REST principles, this project’s API facilitates CRUD (Create, Read, Update, Delete) operations on inventory records. Each API endpoint is designed to perform a specific function, allowing users to interact with the inventory data in an organized, predictable way. This approach provides scalability and reduces the complexity of application maintenance. RESTful APIs are language-agnostic, making them an ideal choice for a wide range of client applications, from web to mobile platforms. This adaptability ensures that the API can be integrated into multiple systems, facilitating seamless data flow across different platforms.

With this project, businesses can achieve accurate, real-time tracking of inventory items, minimize errors, and optimize resource allocation. It stands as a testament to the power of modern software development practices, combining the strengths of RESTful API architecture, MongoDB’s scalability, and a dynamic frontend interface. As inventory remains a key asset for many businesses, the implementation of this project offers a substantial improvement in managing resources effectively and improving operational efficiency in a competitive market landscape.

CHAPTER-2

SYSTEM ARCHITECTURE

The system architecture of the "Build a RESTful API for an Inventory Management System" project is carefully crafted to be modular, scalable, and efficient, ensuring streamlined communication between various components while delivering robust performance for managing inventory data. This architectural design aims to meet the demands of modern applications, where high levels of data integrity, usability, and adaptability are essential. By organizing the system into clearly defined layers, each with its own specific function, the architecture enables seamless integration and interaction among different modules.

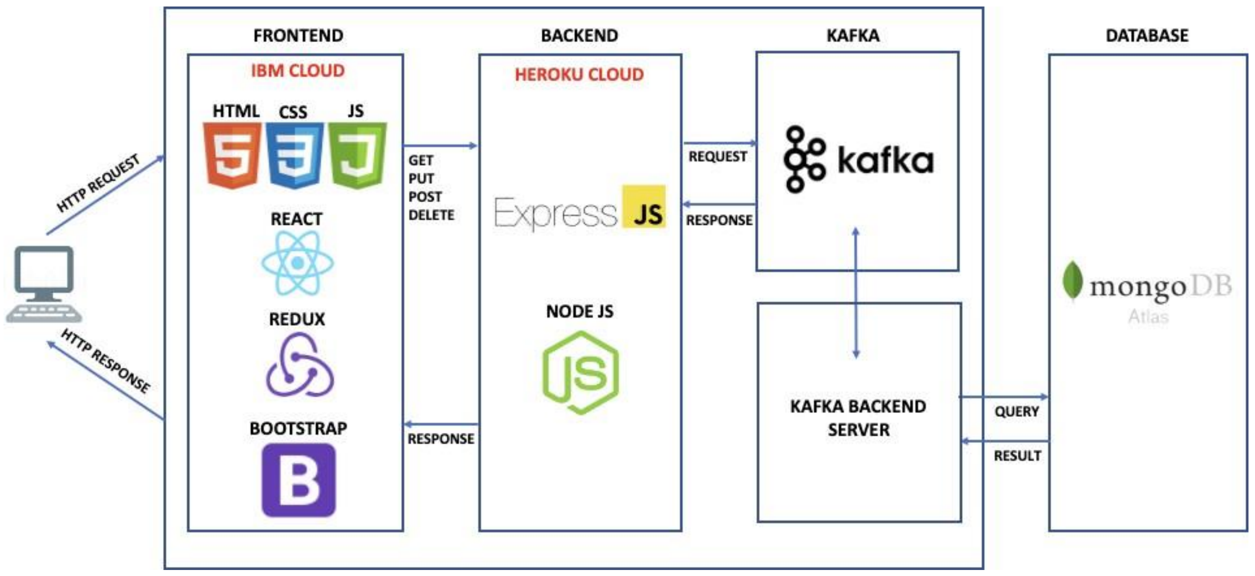


Fig 2.1.1 **System Architecture**

This modularity allows developers to isolate and maintain each component independently, reducing complexity and improving the ease of development and testing. For example, the frontend layer, which comprises HTML, CSS, and JavaScript, is dedicated solely to user interaction, ensuring a responsive and dynamic interface for viewing, adding, updating, and deleting inventory items.

CHAPTER-3

FRONTEND LAYER

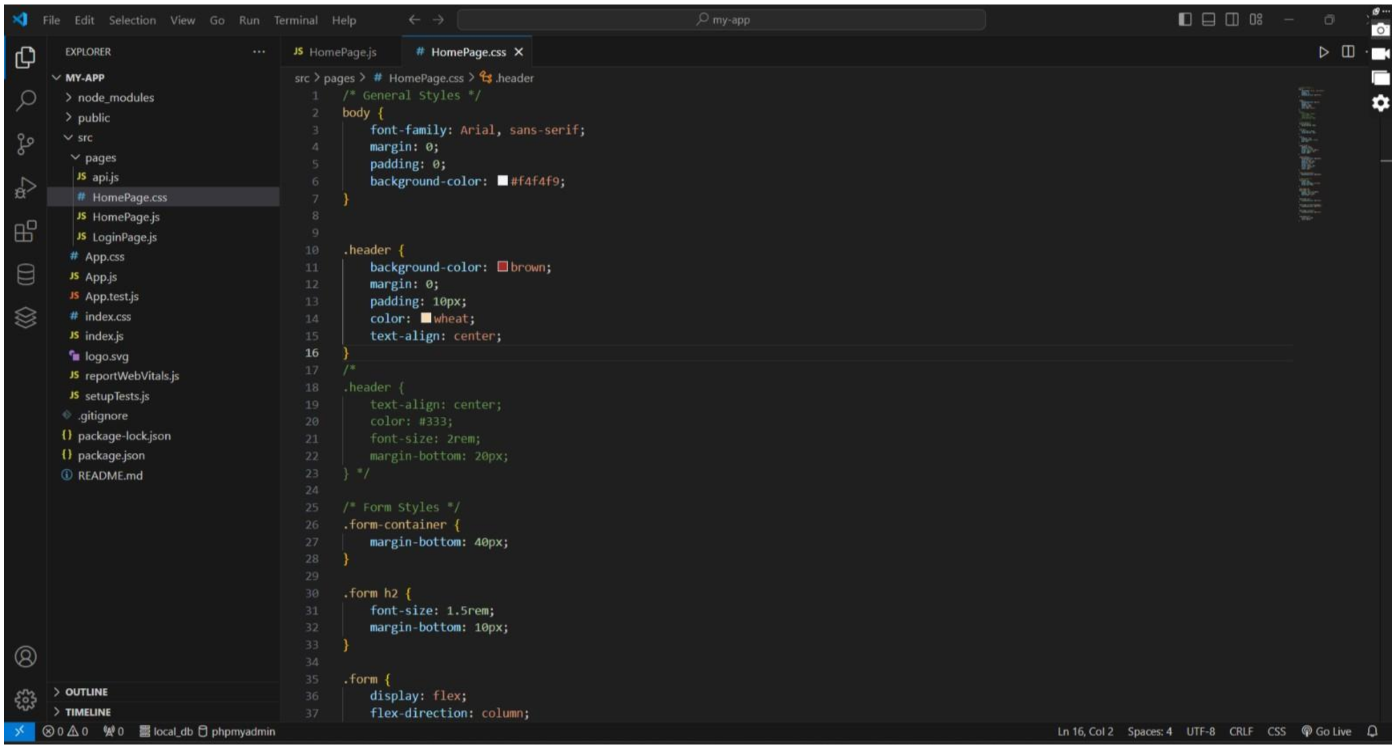
The frontend is the client-facing component of the architecture, responsible for interacting with users and displaying inventory data. Built using HTML, CSS, and JavaScript, this layer is designed for responsiveness, usability, and efficient data handling.

3.1 HTML & CSS

In the “Build a RESTful API for an Inventory Management System” project, HTML and CSS work in tandem to create a structured, user-friendly, and visually appealing interface that enhances the user experience while managing inventory data. HTML (Hyper Text Markup Language) forms the foundation of the user interface by defining the structure and layout of each web page. HTML elements like forms, tables, buttons, and input fields provide essential building blocks that allow users to interact with the system

Fig 3.1.1 **HTML & CSS**

CSS (Cascading Style Sheets), on the other hand, enhances the presentation and layout of these HTML elements, making the application aesthetically pleasing and intuitive to use. CSS provides styling and formatting rules that bring consistency and visual hierarchy to the interface. With CSS, elements like tables, buttons, and forms can be customized with colors, font styles, and spacing, ensuring they are visually distinct and easy to identify.



```css
/* General Styles */
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #f4f4f9;
}


.header {
    background-color: brown;
    margin: 0;
    padding: 10px;
    color: wheat;
    text-align: center;
}
/*
.header {
    text-align: center;
    color: #333;
    font-size: 2rem;
    margin-bottom: 20px;
} */

/* Form Styles */
.form-container {
    margin-bottom: 40px;
}

.form h2 {
    font-size: 1.5rem;
    margin-bottom: 10px;
}

.form {
    display: flex;
    flex-direction: column;
```

Fig 3.1.2 **CSS Code Execution**

CSS enables responsive design, allowing the layout to adapt to different screen sizes, making the application accessible on desktops, tablets, and smartphones. Additionally, CSS frameworks like Bootstrap can be incorporated to quickly implement standardized styling, making the design process faster and ensuring a professional look.

3.2 JAVASCRIPT

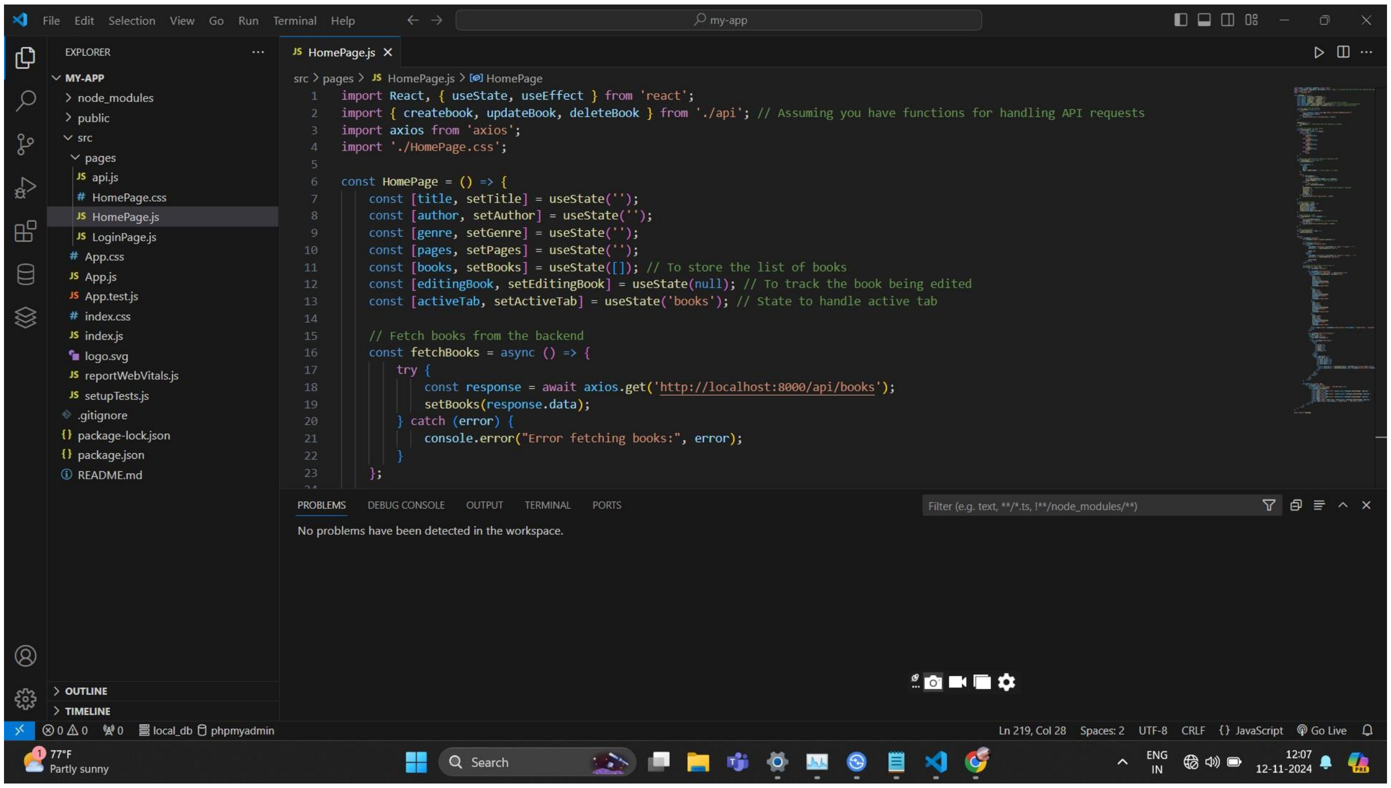
Positioned as the primary scripting language on the frontend, JavaScript is responsible for managing the communication between the user interface (built with HTML and styled by CSS) and the backend RESTful API, allowing users to interact with the system seamlessly. JavaScript's primary function in this project is to handle AJAX (Asynchronous JavaScript and XML) requests, enabling asynchronous communication with the backend API. This means that when a user performs an action, such as adding a new item, updating inventory details, or deleting an item, JavaScript can send an HTTP request to the backend without requiring a full page reload, keeping the interface responsive and user-friendly.

Fig 3.2.1 **JavaScript**

This asynchronous communication is vital for providing a real-time, interactive experience. In addition to handling data fetching and display, JavaScript enables interactive elements that enhance usability. It powers actions like hover effects, button clicks, and form submissions, making the interface more responsive and intuitive.

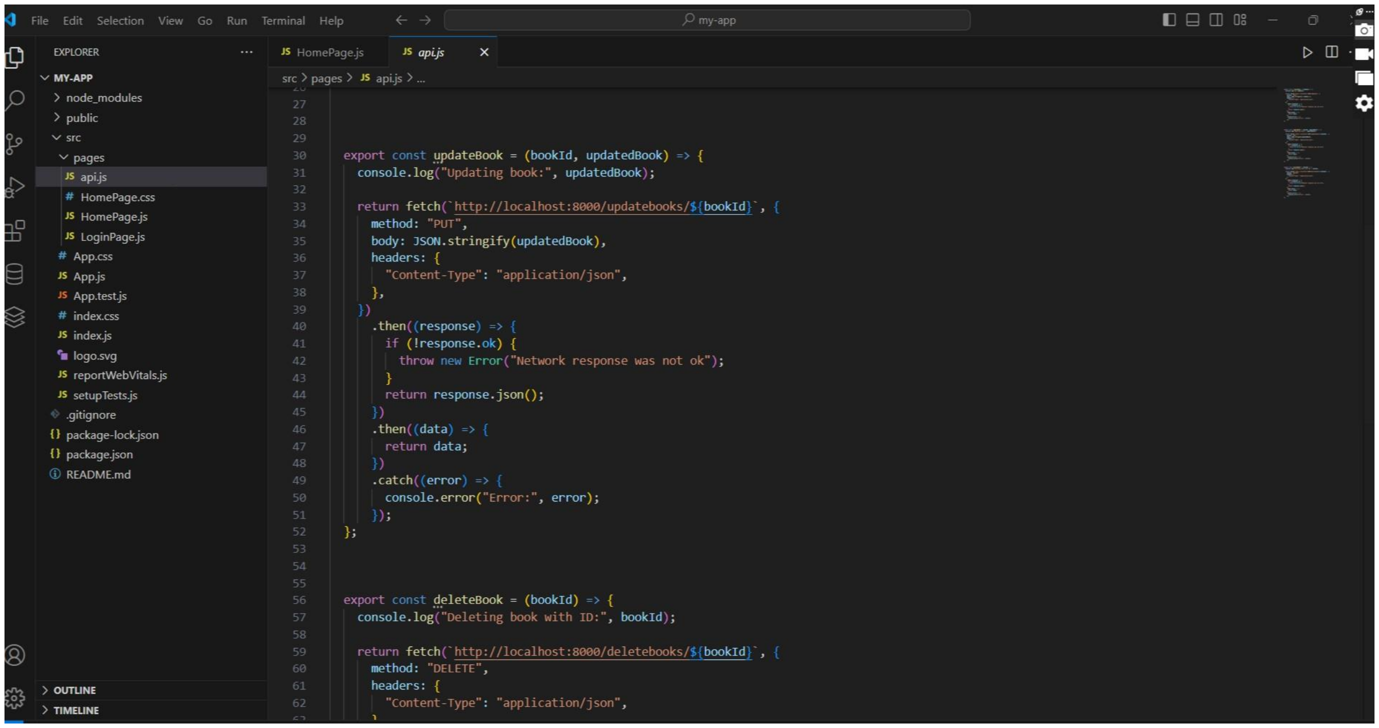
JavaScript's flexibility also supports error handling in API calls, allowing the application to catch errors and provide feedback to the user if something goes wrong, such as network issues or invalid requests. This is particularly useful for displaying meaningful messages, like "Item added successfully" or "Error: Could not delete item." Additionally, JavaScript's interaction with CSS allows for dynamic styling adjustments, like highlighting table rows when selected or changing button colors based on user actions, making the interface feel responsive and alive.

Overall, JavaScript acts as the bridge between the frontend and backend, managing the user interface's dynamic behavior, validating data, handling real-time updates, and ensuring smooth communication with the RESTful API. By integrating with HTML and CSS, JavaScript transforms a static web page into a fully interactive, responsive, and user-centric application, enhancing the efficiency and usability of the inventory management system. This extensive use of JavaScript ensures that users can intuitively and efficiently manage inventory data without technical complexity, making it a critical component in delivering a smooth and satisfying experience.



```javascript
import React, { useState, useEffect } from 'react';
import { createbook, updateBook, deleteBook } from './api'; // Assuming you have functions for handling API requests
import axios from 'axios';
import './HomePage.css';

const HomePage = () => {
  const [title, setTitle] = useState('');
  const [author, setAuthor] = useState('');
  const [genre, setGenre] = useState('');
  const [pages, setPages] = useState('');
  const [books, setBooks] = useState([]); // To store the list of books
  const [editingBook, setEditingBook] = useState(null); // To track the book being edited
  const [activeTab, setActiveTab] = useState('books'); // State to handle active tab

  // Fetch books from the backend
  const fetchBooks = async () => {
    try {
      const response = await axios.get('http://localhost:8000/api/books');
      setBooks(response.data);
    } catch (error) {
      console.error("Error fetching books:", error);
    }
  };
```

No problems have been detected in the workspace.



```javascript



export const updateBook = (bookId, updatedBook) => {
  console.log("Updating book:", updatedBook);

  return fetch(`http://localhost:8000/updatebooks/${bookId}`, {
    method: "PUT",
    body: JSON.stringify(updatedBook),
    headers: {
      "Content-Type": "application/json",
    },
  })
    .then((response) => {
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      return response.json();
    })
    .then((data) => {
      return data;
    })
    .catch((error) => {
      console.error("Error:", error);
    });
};



export const deleteBook = (bookId) => {
  console.log("Deleting book with ID:", bookId);

  return fetch(`http://localhost:8000/deletebooks/${bookId}`, {
    method: "DELETE",
    headers: {
      "Content-Type": "application/json",
```

Fig 3.2.2 **JavaScript code Execution 2**

CHAPTER-4

API LAYER (BACKEND)

API layer functions as the core communication hub between the frontend interface and the backend database, playing a crucial role in managing and facilitating the flow of data within the system. This layer is designed as a RESTful API, adhering to REST (Representational State Transfer) principles, which makes it stateless, scalable, and easy to understand and use. The primary role of the API layer is to provide a set of well-defined endpoints through which the frontend can interact with the system's data. Using standard HTTP methods such as GET, POST, PUT, and DELETE, each endpoint in the API corresponds to a specific action: retrieving data, adding new entries, updating existing records, or removing inventory items from the database. This organized approach enables clear, structured communication between the frontend and backend, which is essential for seamless user experience and operational efficiency.

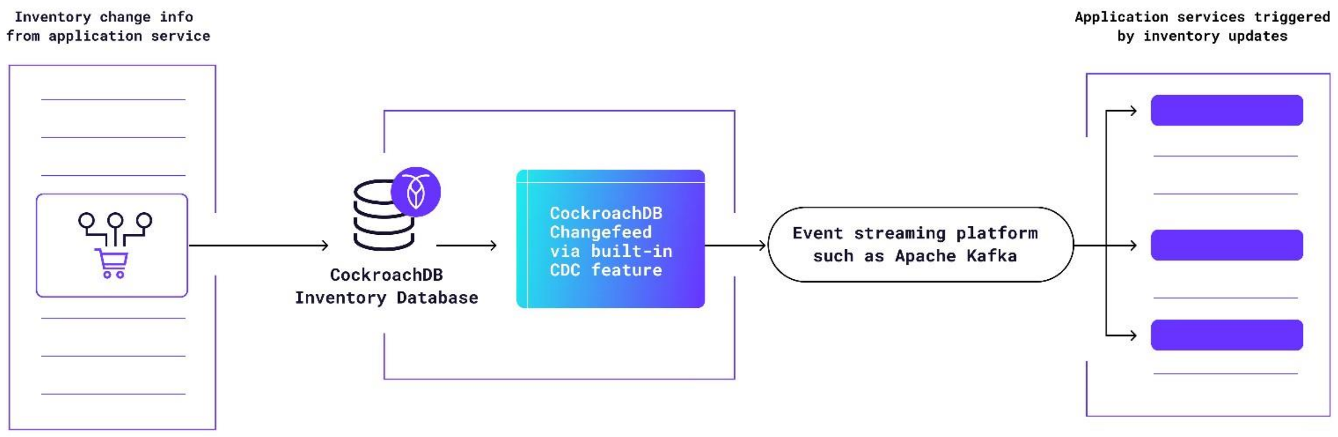
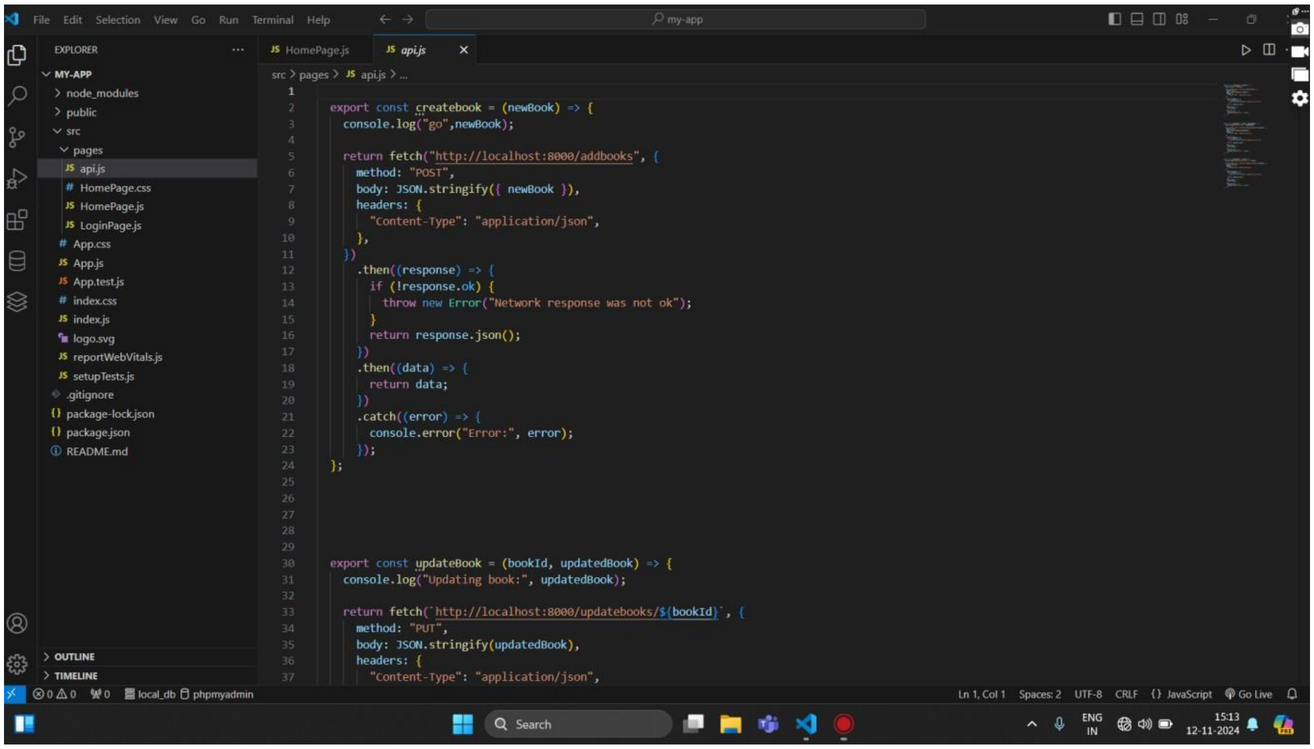
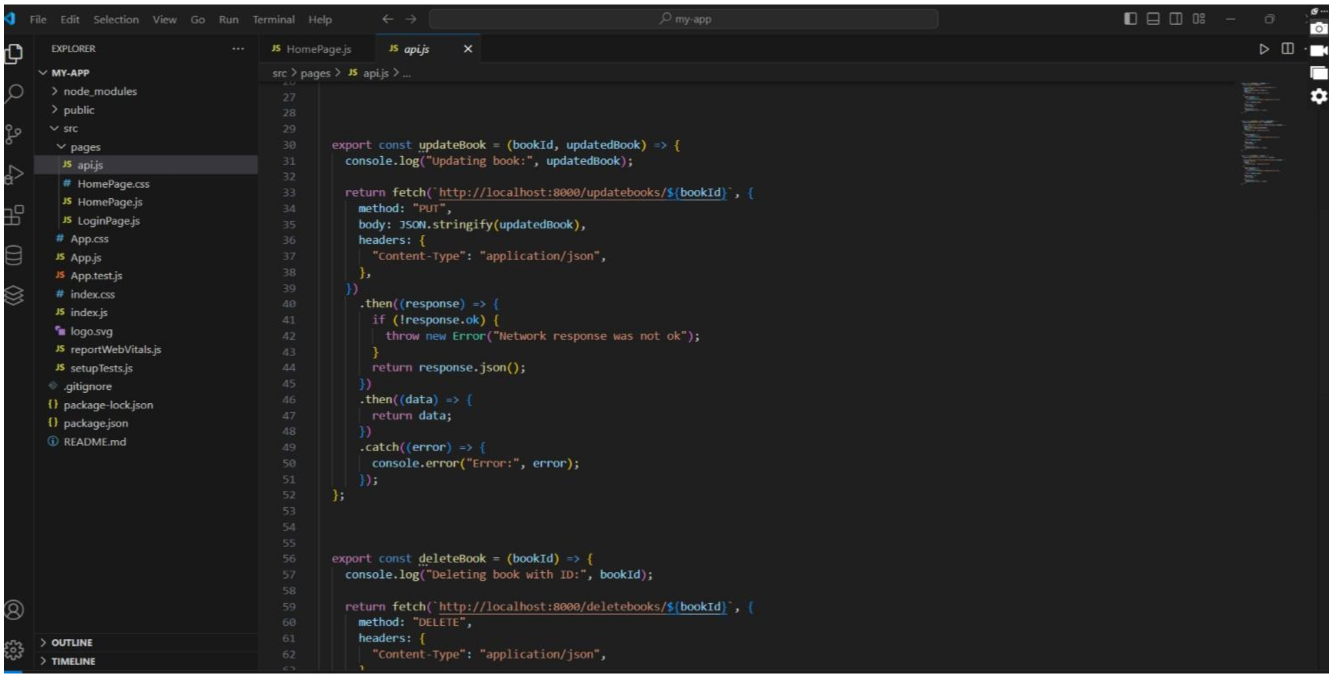

Inventory change info from application service
CockroachDB Inventory Database
CockroachDB Changefeed via built-in CDC feature
Event streaming platform such as Apache Kafka
Application services triggered by inventory updates

Fig 4.1.1 **RESTful API for Inventory Management**

When a user performs an action on the frontend—such as viewing the inventory, adding a new item, editing details of an existing product, or deleting an item—the API layer processes these requests by translating them into appropriate database operations. The API layer interacts with MongoDB, the database chosen for this project, allowing for flexible and efficient data storage and retrieval.



```

export const createbook = (newBook) => {
  console.log("go",newBook);

  return fetch("http://localhost:8000/addbooks", {
    method: "POST",
    body: JSON.stringify({ newBook }),
    headers: {
      "Content-Type": "application/json",
    },
  })
    .then((response) => {
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      return response.json();
    })
    .then((data) => {
      return data;
    })
    .catch((error) => {
      console.error("Error:", error);
    });
};





export const updateBook = (bookId, updatedBook) => {
  console.log("Updating book:", updatedBook);

  return fetch(`http://localhost:8000/updatebooks/${bookId}`, {
    method: "PUT",
    body: JSON.stringify(updatedBook),
    headers: {
      "Content-Type": "application/json",
```



```



export const updateBook = (bookId, updatedBook) => {
  console.log("Updating book:", updatedBook);

  return fetch(`http://localhost:8000/updatebooks/${bookId}`, {
    method: "PUT",
    body: JSON.stringify(updatedBook),
    headers: {
      "Content-Type": "application/json",
    },
  })
    .then((response) => {
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      return response.json();
    })
    .then((data) => {
      return data;
    })
    .catch((error) => {
      console.error("Error:", error);
    });
};



export const deleteBook = (bookId) => {
  console.log("Deleting book with ID:", bookId);

  return fetch(`http://localhost:8000/deletebooks/${bookId}`, {
    method: "DELETE",
    headers: {
      "Content-Type": "application/json",
```

Fig 4.1.2 **API for the Inventory**

JSON responses are then parsed on the frontend and rendered in a user-friendly format for display, such as in a table view, providing users with a clear and structured overview of their inventory. Lastly, the API layer's role in security is indispensable. Security measures, such as authentication and authorization, can be implemented to control access to sensitive inventory data and prevent unauthorized users from performing certain actions. Token-based authentication, such as JSON Web Tokens (JWT), can be integrated to verify users' identities and limit access to only those with valid credentials. This ensures that only authorized personnel can modify inventory records, protecting the data from unauthorized access and manipulation.

4.1 NODE.JS

Node.js plays a critical role in the API layer by serving as the backend runtime environment that powers the server-side logic. Node.js, an open-source, JavaScript-based runtime, is built on Chrome's V8 JavaScript engine and is designed for high-performance, non-blocking I/O operations, making it ideal for handling the numerous simultaneous client requests an inventory management system may receive. This efficiency is particularly valuable for our project, as Node.js enables the API to manage real-time data transactions, ensuring that users can add, retrieve, update, or delete inventory items with minimal delay. Its event-driven architecture allows the API layer to handle multiple requests concurrently, which is essential for scalability and responsiveness, as the system needs to handle growing volumes of inventory data and user requests without sacrificing performance.

Fig 4.1.3 **Node.js**

4.2 HTTP METHODS

In the "Build a RESTful API for an Inventory Management System" project, several HTTP methods and endpoints are involved in facilitating communication between the frontend and the backend. These methods—GET, POST, PUT, DELETE—are essential for enabling users to interact with inventory data effectively through the API.

- **GET Method**: The **GET** method is used to retrieve data from the server. The endpoint for retrieving all inventory items might be **/api/items,** which returns a list of all products in the inventory. A more specific GET request, such as **/api/items/{id},** retrieves data for a single inventory item identified by its unique ID.

- **POST Method**: The **POST** method is used for creating new data on the server. The endpoint, such as **/api/items,** is designed to accept data (like item name, quantity, price, etc.) sent from the frontend. When a POST request is made to this endpoint, the API processes the request, validates the data, and stores it in the database (MongoDB).

- **PUT Method**: The **PUT** method is employed for updating existing resources on the server. For instance, the endpoint **/api/items/{id}** allows users to update specific fields of an item (such as quantity or price) by sending updated data in the request body.

- **DELETE Method**: The **DELETE** method is responsible for removing resources from the server. The endpoint **/api/items/{id}** would delete the item identified by the unique ID provided in the URL.

Along with these core **CRUD** (Create, Read, Update, Delete) operations, additional endpoints might be designed for specialized actions. For example, a **GET** endpoint such as **/api/items/search** could be used to search for inventory items based on specific criteria like name or category. Similarly, **POST** endpoints for user authentication **(e.g., /api/auth/login)** might be implemented to handle user logins, ensuring that only authorized personnel can access certain endpoints.

CHAPTER-5

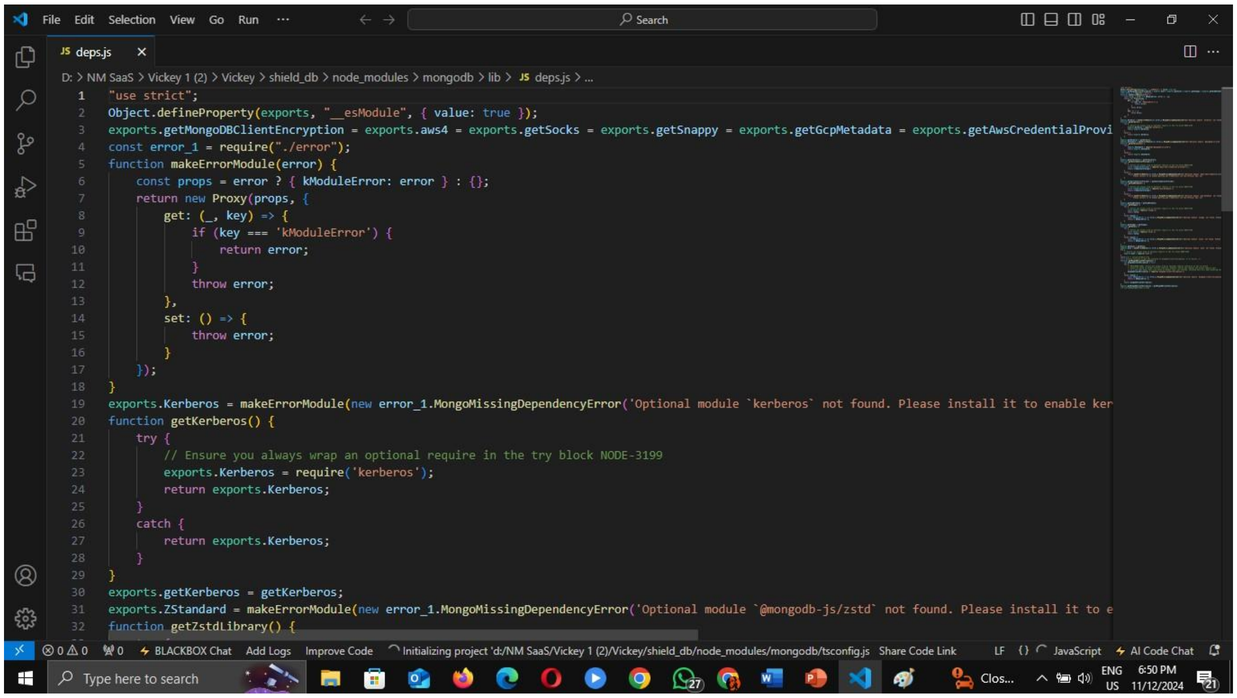
DATABASE LAYER

The database layer plays a crucial role in storing, managing, and retrieving inventory data efficiently. MongoDB, a NoSQL database, is utilized as the primary data storage solution in this project. MongoDB is chosen due to its flexibility, scalability, and ease of integration with Node.js. Unlike traditional relational databases, MongoDB stores data in JSON-like documents, known as BSON (Binary JSON), which makes it highly adaptable to the dynamic and evolving nature of the inventory system.

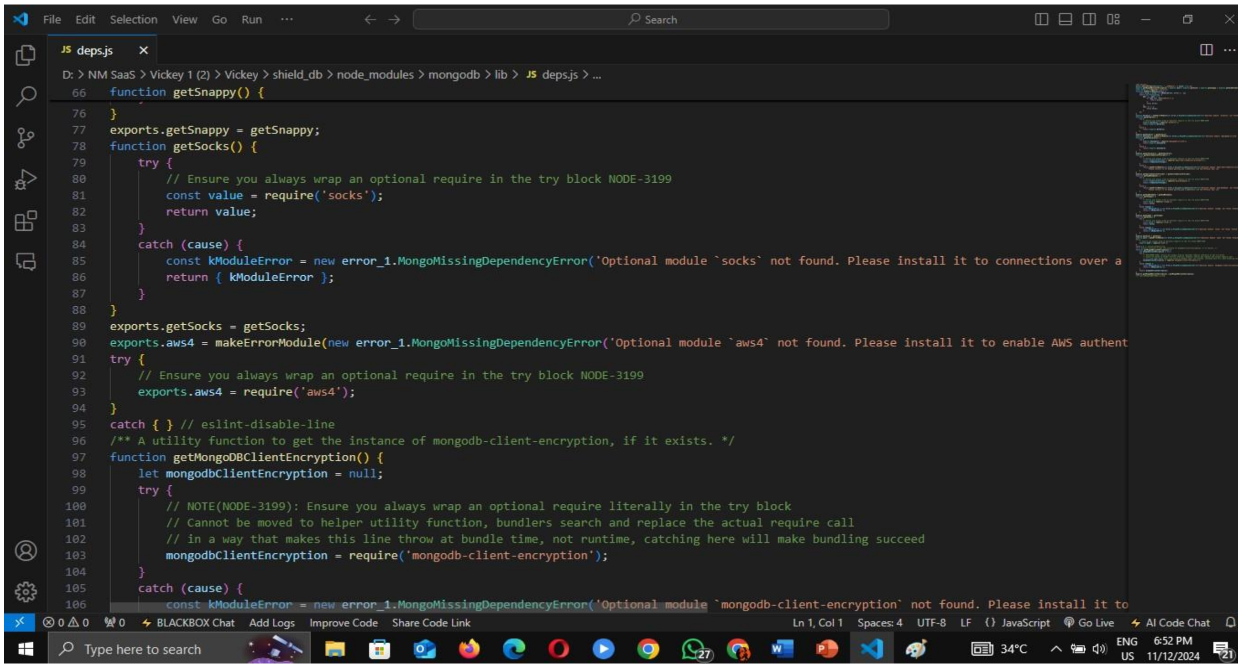
5.1 MONGO DB

In this system, the **MongoDB** database stores inventory items as documents within a collection. Each document represents an individual item and contains fields such as itemName, itemCode, quantity, price, description, and category, among others. This document structure is flexible, meaning that new fields can be added easily without disrupting the existing data. For instance, if new attributes, such as "supplier" or "expiration date," need to be tracked for inventory items, they can be added to the relevant documents without requiring complex schema changes or database migrations, a feature that is a significant advantage over relational databases.

Fig 5.1.1 **Mongo DB**



```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.getMongoDBClientEncryption = exports.aws4 = exports.getSocks = exports.getSnappy = exports.getGcpMetadata = exports.getAwsCredentialProvi
const error_1 = require("./error");
function makeErrorModule(error) {
    const props = error ? { kModuleError: error } : {};
    return new Proxy(props, {
        get: (_, key) => {
            if (key === 'kModuleError') {
                return error;
            }
            throw error;
        },
        set: () => {
            throw error;
        }
    });
}
exports.Kerberos = makeErrorModule(new error_1.MongoMissingDependencyError('Optional module `kerberos` not found. Please install it to enable ker
function getKerberos() {
    try {
        // Ensure you always wrap an optional require in the try block NODE-3199
        exports.Kerberos = require('kerberos');
        return exports.Kerberos;
    }
    catch {
        return exports.Kerberos;
    }
}
exports.getKerberos = getKerberos;
exports.ZStandard = makeErrorModule(new error_1.MongoMissingDependencyError('Optional module `@mongodb-js/zstd` not found. Please install it to e
function getZstdLibrary() {
```



```
function getSnappy() {
}
exports.getSnappy = getSnappy;
function getSocks() {
    try {
        // Ensure you always wrap an optional require in the try block NODE-3199
        const value = require('socks');
        return value;
    }
    catch (cause) {
        const kModuleError = new error_1.MongoMissingDependencyError('Optional module `socks` not found. Please install it to connections over a
        return { kModuleError };
    }
}
exports.getSocks = getSocks;
exports.aws4 = makeErrorModule(new error_1.MongoMissingDependencyError('Optional module `aws4` not found. Please install it to enable AWS authent
try {
    // Ensure you always wrap an optional require in the try block NODE-3199
    exports.aws4 = require('aws4');
}
catch { } // eslint-disable-line
/** A utility function to get the instance of mongodb-client-encryption, if it exists. */
function getMongoDBClientEncryption() {
    let mongodbClientEncryption = null;
    try {
        // NOTE(NODE-3199): Ensure you always wrap an optional require literally in the try block
        // Cannot be moved to helper utility function, bundlers search and replace the actual require call
        // in a way that makes this line throw at bundle time, not runtime, catching here will make bundling succeed
        mongodbClientEncryption = require('mongodb-client-encryption');
    }
    catch (cause) {
        const kModuleError = new error_1.MongoMissingDependencyError('Optional module `mongodb-client-encryption` not found. Please install it to
```

Fig 5.1.2 **Mongo DB Library Install**

CHAPTER-6

SYSTEM FLOW

The system flow of the "Build a RESTful API for an Inventory Management System" project outlines the sequential interactions between the different components—frontend, backend, database, and client—to ensure smooth data processing and user experience. This detailed flow helps to define how the system processes requests from users, handles data, and responds to various operations in the inventory system.

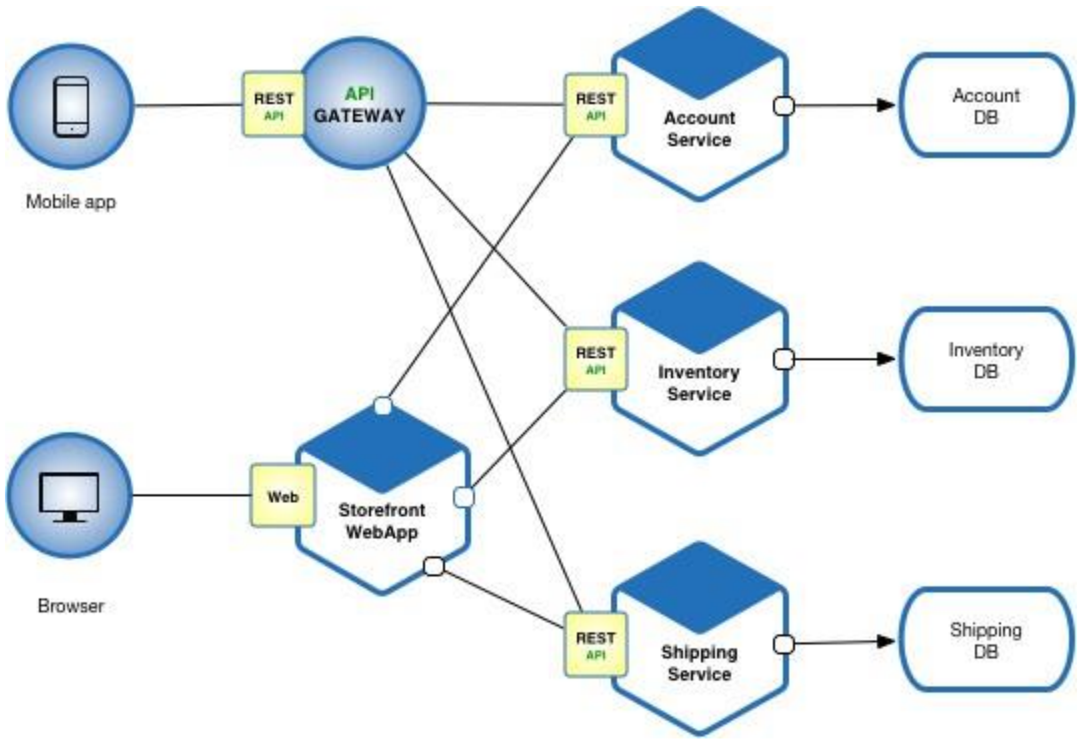


Fig 6.1.1 **System Flow**

- **Client Interaction**: The system flow begins when a user (client) interacts with the frontend interface, built using HTML, CSS, and JavaScript. The frontend provides a user-friendly platform where clients can view, add, update, or delete inventory items.

- **Frontend Requests**: Once the client initiates an action, the frontend JavaScript code (using AJAX or Fetch API) sends a corresponding HTTP request to the backend RESTful API. The request type varies based on the action.

- **API Layer**: The RESTful API, implemented in Node.js with Express, acts as the central processing layer of the system. When it receives an HTTP request, the API routes it to the appropriate endpoint.

- **Data Validation and Business Logic**: Before data is sent to the database, the API layer includes validation logic to maintain data integrity. This validation may use libraries like Joi or Mongoose validators, particularly with MongoDB, to ensure the incoming data is accurate and conforms to the system's rules.

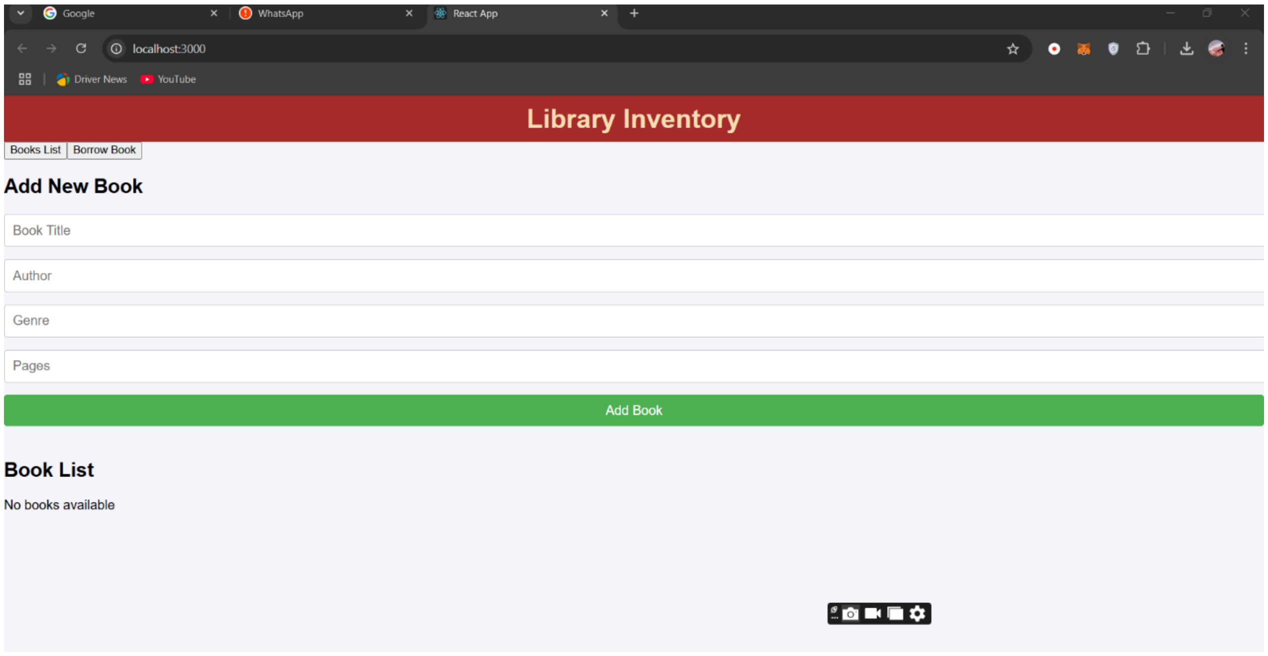
- **Database Interaction (MongoDB)**: After validation, the API layer interacts with the MongoDB database to perform the requested operation. MongoDB, a NoSQL database, stores inventory data in BSON (Binary JSON) format, which aligns naturally with JSON data used by the API.

- **Real-Time Feedback and User Notifications**: If the project incorporates real-time updates, such as WebSockets or change streams in MongoDB, users may see immediate updates on the frontend for actions like additions, deletions, or modifications of items by other users.

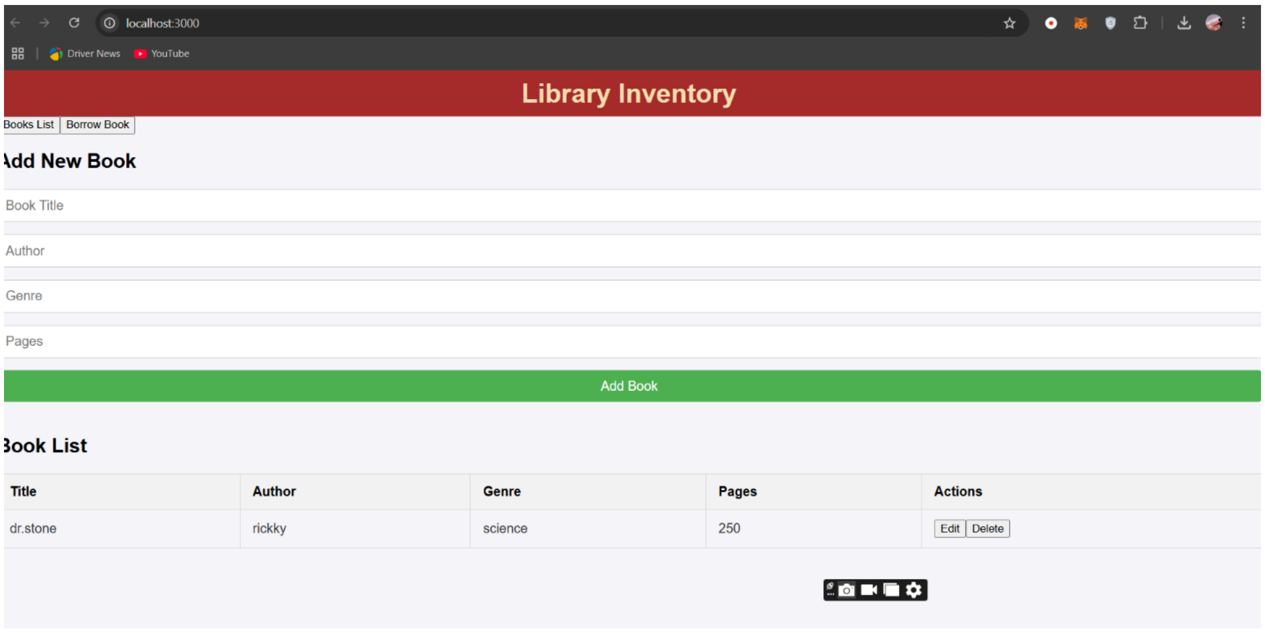
- **Logging and Analytics**: Throughout the system flow, logs are generated at various stages—API requests, database interactions, and errors.

CHAPTER-7

INTERFACE OF THE INVENTORY (O/P)

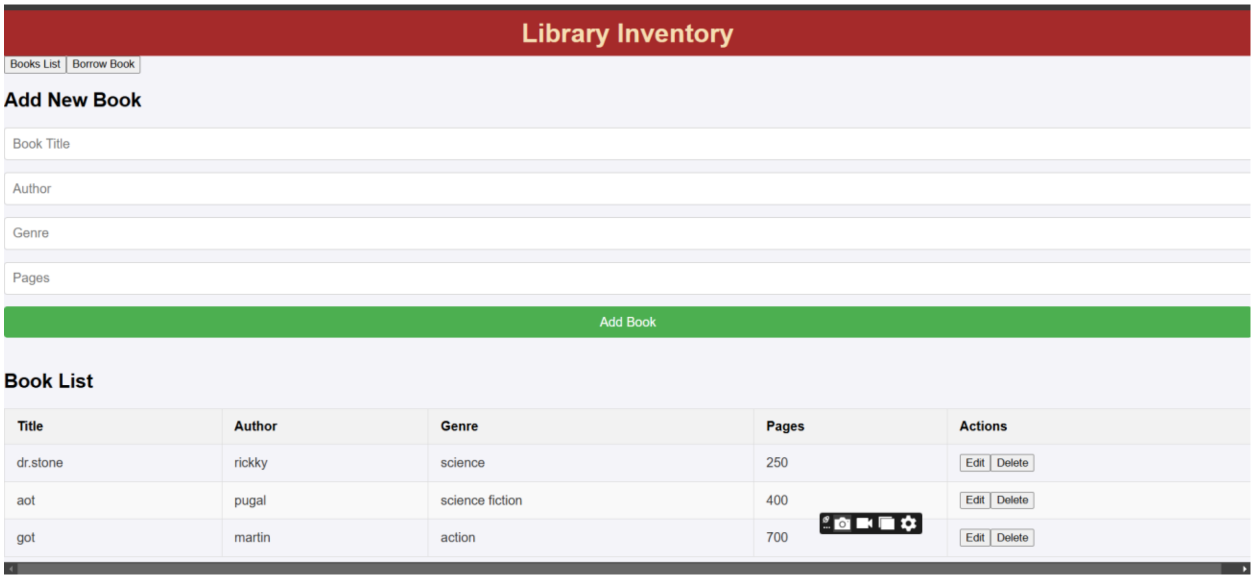


No books available

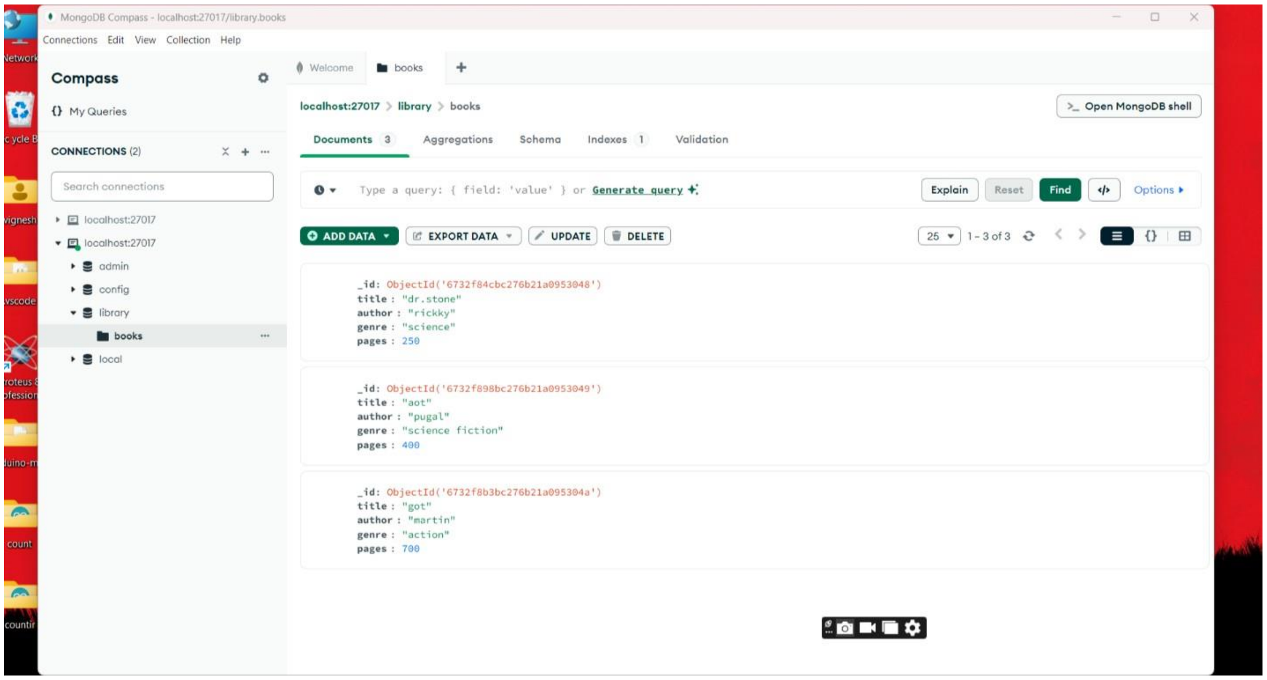


Title	Author	Genre	Pages	Actions
dr.stone	rickky	science	250	Edit Delete

Fig 7.1.1 **Interface of Inventory Mgt System**

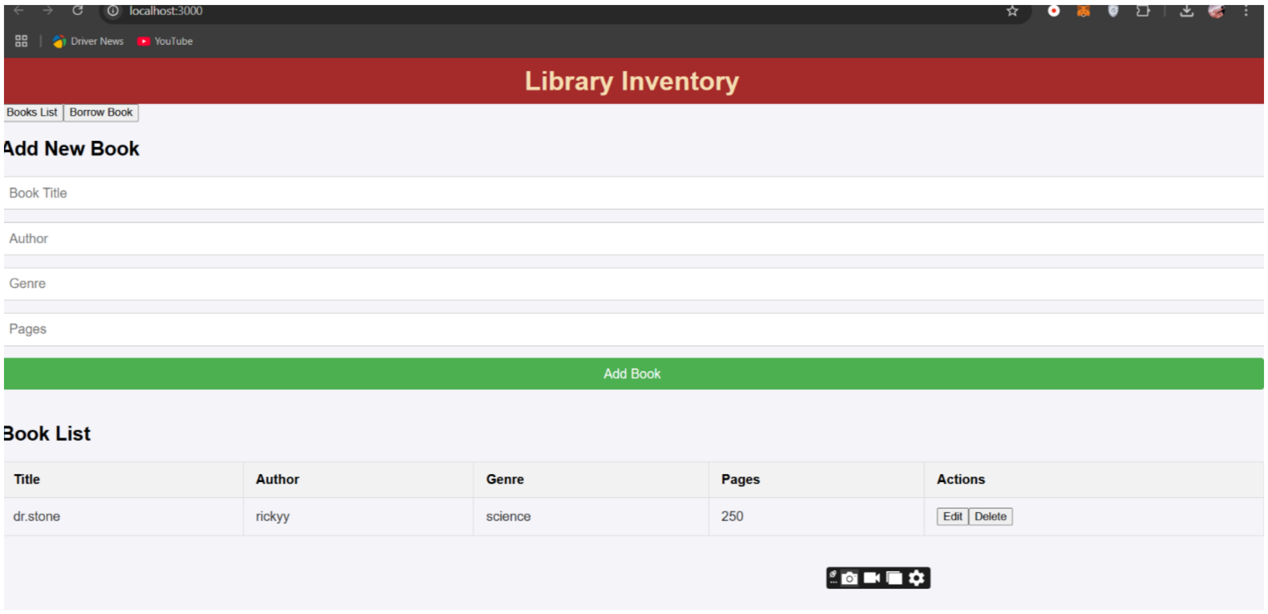


Title	Author	Genre	Pages	Actions
dr.stone	rickky	science	250	Edit Delete
aot	pugal	science fiction	400	Edit Delete
got	martin	action	700	Edit Delete



```
_id: ObjectId('6732f84cbc276b21a0953048')
title : "dr.stone"
author : "rickky"
genre : "science"
pages : 250

_id: ObjectId('6732f898bc276b21a0953049')
title : "aot"
author : "pugal"
genre : "science fiction"
pages : 400

_id: ObjectId('6732f8b3bc276b21a095304a')
title : "got"
author : "martin"
genre : "action"
pages : 700
```

Fig 7.1.2 **Adding Books to the Inventory**



Add New Book

Book Title

Author

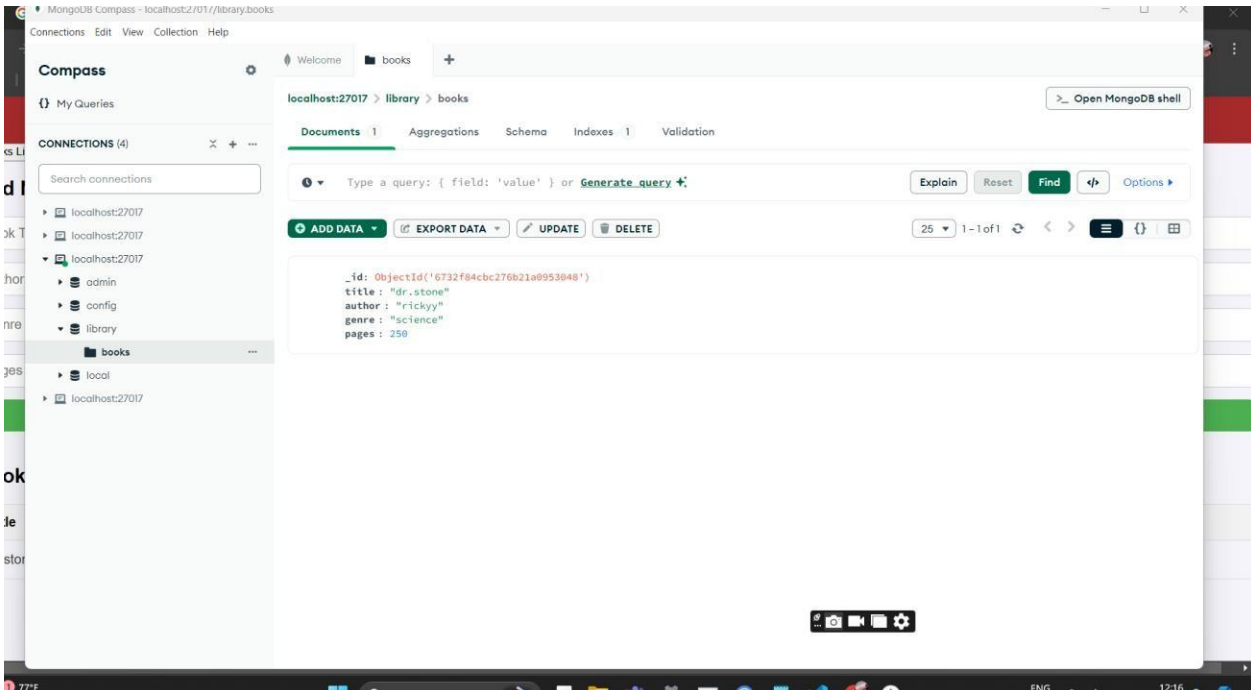
Genre

Pages

Add Book

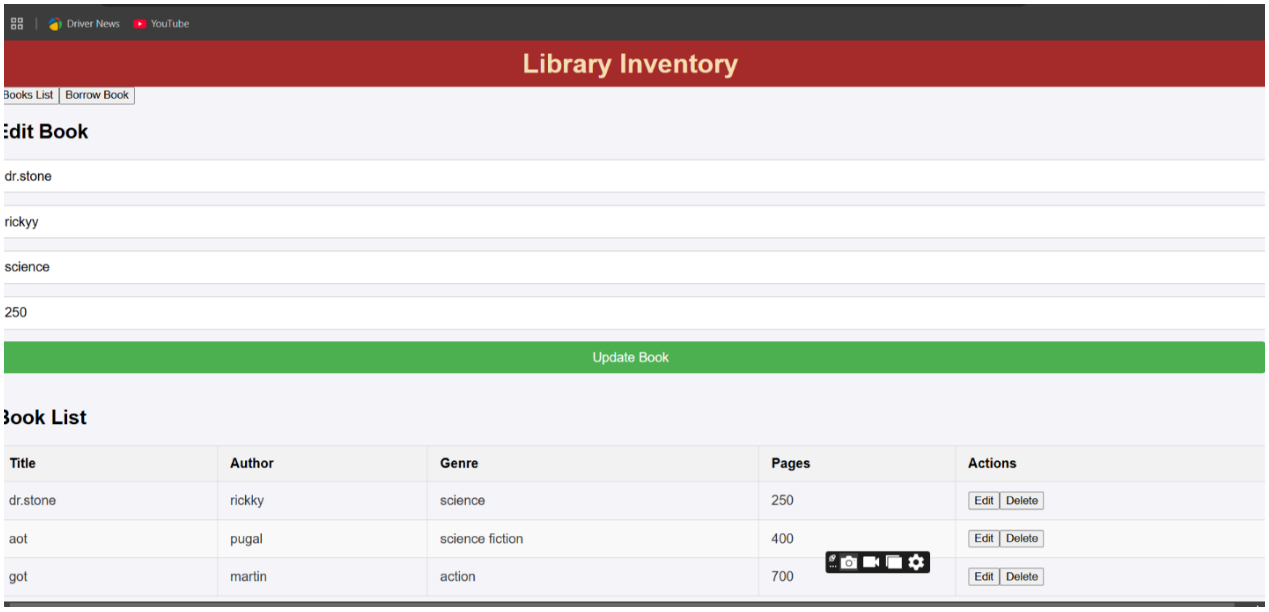
Book List

Title	Author	Genre	Pages	Actions
dr.stone	rickyy	science	250	Edit Delete



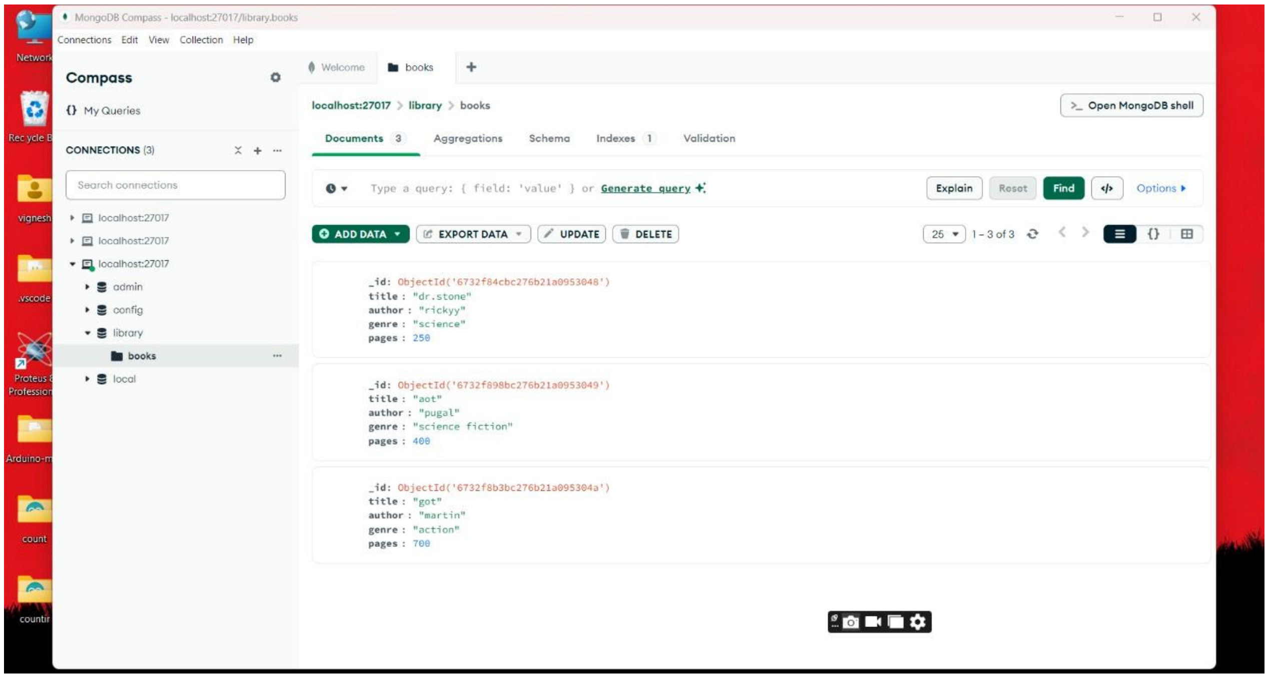
```
_id: ObjectId('6732f84cbc276b21a0953048')
title : "dr.stone"
author : "rickyy"
genre : "science"
pages : 250
```

Fig 7.1.3 **Deleting a book from Inventory**



dr.stone
rickyy
science
250

Title	Author	Genre	Pages	Actions
dr.stone	rickky	science	250	Edit Delete
aot	pugal	science fiction	400	Edit Delete
got	martin	action	700	Edit Delete



```
_id: ObjectId('6732f84cbc276b21a0953048')
title : "dr.stone"
author : "rickyy"
genre : "science"
pages : 250

_id: ObjectId('6732f898bc276b21a0953049')
title : "aot"
author : "pugal"
genre : "science fiction"
pages : 400

_id: ObjectId('6732f8b3bc276b21a095304a')
title : "got"
author : "martin"
genre : "action"
pages : 700
```

Fig 7.14 **Updating the Library Inventory**

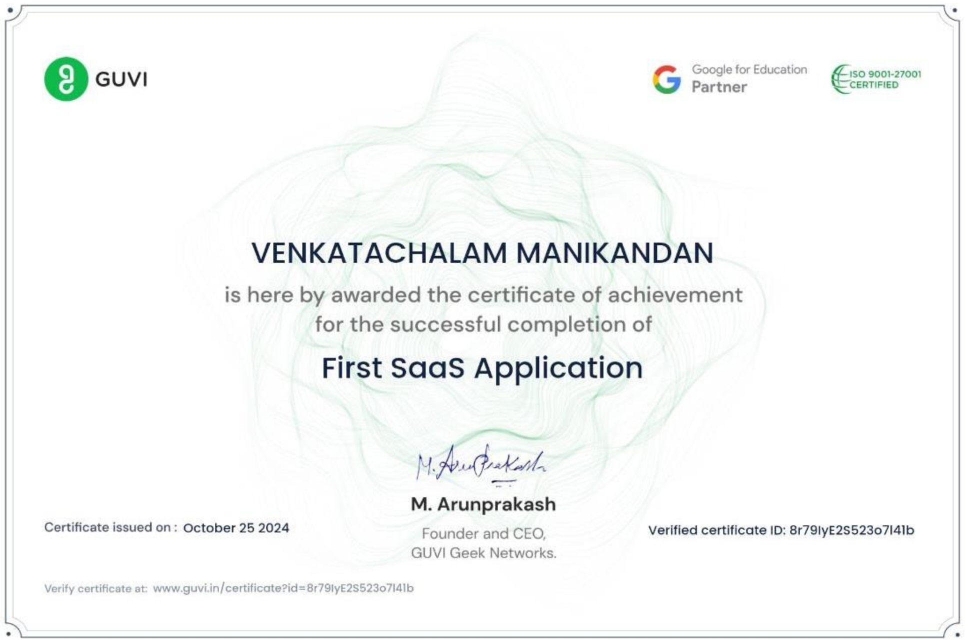
CHAPTER-8

CONCLUSION

The "Build a RESTful API for an Inventory Management System" project serves as an efficient, scalable, and modern solution for managing inventory data with robust technologies and a carefully designed architecture. By combining the strengths of various programming languages, tools, and frameworks, this project demonstrates the powerful synergy between frontend and backend technologies, allowing for smooth and responsive inventory management. At the core of the project is Node.js, a versatile server-side runtime environment, which works with Express to establish a fast and lightweight RESTful API layer. This API efficiently handles CRUD operations—Create, Read, Update, and Delete—that enable users to interact with inventory data seamlessly and intuitively. The project's RESTful API architecture promotes modularity and separation of concerns, where each layer—frontend, backend, and database—can operate independently yet communicate seamlessly. The use of JSON as a data format for client-server communication ensures consistent data exchange, and the standardization of HTTP methods (GET, POST, PUT, DELETE) aligns with widely accepted REST principles. This modular approach also enables the system to be compatible with various platforms and devices, offering cross-platform flexibility. Security measures, including authentication protocols like JSON Web Tokens (JWT), ensure that only authorized users can perform sensitive operations, protecting data integrity and preventing unauthorized access.

In conclusion, this project successfully integrates Node.js, Express, MongoDB, HTML, CSS, and JavaScript to deliver a powerful, secure, and responsive inventory management system that meets the needs of users while allowing for future expansion and adaptability. Its RESTful architecture supports easy maintenance and potential integrations with other systems, making it versatile and scalable. Through the strategic use of these technologies, the project has achieved a modular, efficient, and highly functional solution for inventory management, demonstrating the value of combining server-side robustness, flexible data handling, and user-friendly frontend design.

CERTIFICATES



GUVI

Google for Education Partner

ISO 9001-27001 CERTIFIED

VENKATACHALAM MANIKANDAN

is here by awarded the certificate of achievement for the successful completion of

First SaaS Application

[illegible]

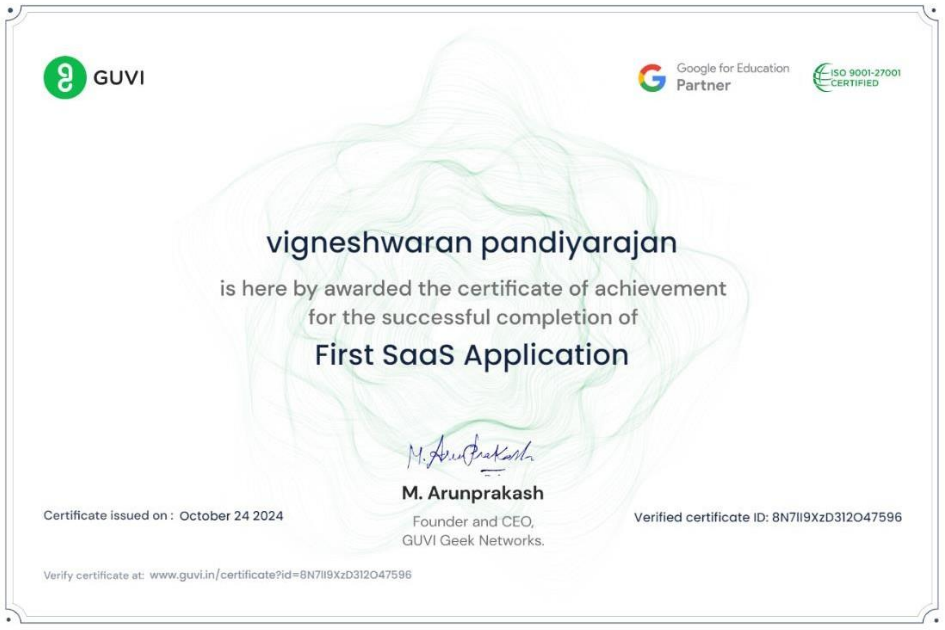
M. Arunprakash

Founder and CEO,
GUVI Geek Networks.

Certificate issued on : October 25 2024

Verified certificate ID: 8r79IyE2S523o7I41b

Verify certificate at: www.guvi.in/certificate?id=8r79IyE2S523o7I41b



GUVI

Google for Education Partner

ISO 9001-27001 CERTIFIED

vigneshwaran pandiyarajan

is here by awarded the certificate of achievement for the successful completion of

First SaaS Application

[illegible]

M. Arunprakash

Founder and CEO,
GUVI Geek Networks.

Certificate issued on : October 24 2024

Verified certificate ID: 8N7II9XzD312O47596

Verify certificate at: www.guvi.in/certificate?id=8N7II9XzD312O47596