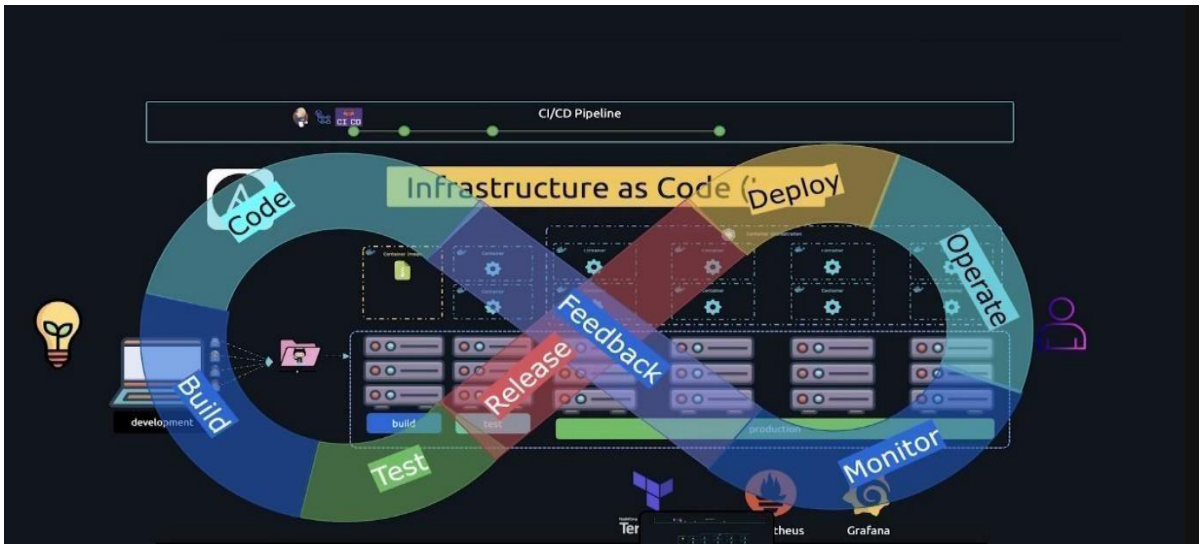
International Institute of Information Technology, Bangalore

CS 816- Software Production Engineering – Major Project



Library Management System

Developed By

Lavish Sainik (MT2023183)

Tezaswa Awasthi (MT2023184)

Table of Contents

1 Abstract

Library Management System is a full-stack web application designed to facilitate efficient management of a library's book collection. The application provides a platform for users to browse, add, remove, and categorize books within the library. It ensures that all visitors can view book information, while authenticated users can manage the book inventory, with additional administrative capabilities for managers. This project leverages DevOps principles to ensure continuous integration and continuous deployment, enabling seamless updates and robust management of the system's infrastructure.

1.1 Problem Definition

Managing a library's extensive collection of books manually is challenging and prone to errors. Issues such as misplaced books, inefficient tracking of borrowed books, and difficulties in maintaining user records can significantly hinder library operations.

1.2 Solution Description

The LibraryManagementSystem addresses these issues by providing an automated and userfriendly platform for library management. The system allows users to register and log in, and provides features for adding, removing, and categorizing books. Managers have additional capabilities to oversee the entire library inventory and user activities.

1.2.1 Features of the Website

User Authentication:

The website provides a secure authentication system allowing library members to create accounts, log in securely, and access personalized features. User credentials are encrypted to ensure data security.

Book Management:
Users can effortlessly manage the library's book inventory through intuitive interfaces. They can add new books to the database, remove existing ones, and update book details as needed. The system ensures seamless book management, reducing manual effort and potential errors.

User Interface:
The website offers dedicated interfaces tailored to meet the needs of both general users and managers. General users have access to features such as browsing books, searching for specific titles, and viewing book details. Managers, on the other hand, have additional administrative functionalities to oversee the entire library inventory, manage user accounts, and moderate content.

Categorization:
Books within the library are organized systematically based on various criteria such as genre, author, publication year, and more. Users can easily browse through different categories to find books of interest. The categorization feature enhances user experience by streamlining the search process and ensuring relevant recommendations.1.3 Important Links

2 Introduction

2.1 What is DevOps?

DevOps is a methodology that integrates software development (Dev) with IT operations (Ops) to enhance the software development lifecycle (SDLC). By fostering collaboration between development and operations teams, DevOps aims to reduce the time to market for software products and ensure continuous delivery and integration, ultimately leading to the production of high-quality software.

2.2 Why to Use DevOps?

Incorporating DevOps into software development has become essential due to its numerous benefits over traditional methods. DevOps enables faster product development cycles, quicker bug fixes, and more straightforward maintenance. This approach supports the rapid deployment of high-quality software, improving overall efficiency and productivity in the development process.

2.3 Tech Stack

- Java
- Spring Boot
- Digital Ocean

- Thymeleaf
- Thymeleaf Fragments
- HTML5
- CSS
- JavaScript
- Spring MVC
- JDBC
- H2 Database Engine (In-memory)
- JUnit test framework
- Spring Security
- Twitter Bootstrap
- Maven

2.4 Operation Tools Used

- **Source Control Management (SCM):** Git and GitHub
- **Continuous Integration (CI):** GitHub Actions
- **Continuous Deployment (CD):** Azure CLI Plugin (in GitHub Actions)
- **Containerization:** Docker
- **Build Tool:** Maven
- **Vulnerability:** Trivy ● **Code Quality:** SonarQube ● **Monitoring:**
    - **Hardware Metrics:** Node Exporter
    - **Availability:** Blackbox Exporter
    - **Visualization:** Grafana
- **Cloud Platform:** Digital Ocean

System Architecture and Infrastructure

Architecture Diagram

3 Backend

3.1 API Overview

The backend of the application is designed to provide robust and scalable services using Java and Spring Boot. The primary focus is on two main functionalities: authentication and student management. These functionalities are exposed through RESTful APIs, which allow for interaction between the client-side application and the server. The APIs ensure secure and efficient handling of user data and student-related operations, adhering to best practices in API design.

AuthenticationController.java

The AuthenticationController.java is a critical component of the backend, responsible for handling all aspects of user authentication. This includes providing endpoints for user login, registration, and token management. The controller uses Spring Security to manage authentication and authorization, ensuring that user data is protected and that only authenticated users can access certain endpoints.

Key functionalities of AuthenticationController.java include:

- **Login Endpoint**: Validates user credentials and generates a JWT token for session management.
- **Registration Endpoint**: Allows new users to register by providing necessary details such as username, password, and email.

```java
@RequestMapping(path = "/api/auth")
public class AuthenticationController {

    private AuthenticationService authenticationService;
     public AuthenticationController(AuthenticationService authenticationService){
        this.authenticationService = authenticationService;
    }

    @PostMapping("/registerStudent")
    public ResponseEntity<String> registerStudent(@RequestBody RegisterRequest registerRequest){
        return new ResponseEntity<>(authenticationService.registerStudent(registerRequest), HttpStatus.OK);
    }

    @PostMapping("/loginStudent")
    public ResponseEntity<LoginResponse> loginStudent(@RequestBody LoginRequest loginRequest) {
        LoginResponse loginStudent = authenticationService.loginStudent(loginRequest);
        HttpStatus status = loginStudent.getId() != null ? HttpStatus.OK : HttpStatus.UNAUTHORIZED;
        return new ResponseEntity<>(loginStudent, status);
    }
}
```

StudentController.java

The StudentController.java manages all student-related operations within the application. This includes adding new students, updating existing student records, and retrieving student information. The controller ensures that all operations are performed securely and efficiently, making use of Spring Data JPA for database interactions.

```
@RestController
@RequestMapping(path = "api/student") public class
StudentController {     private final StudentService
studentService;     private final CategoriesService
categoriesService;


    public StudentController(StudentService studentService,CategoriesService
categoriesService) {         this.studentService = studentService;
this.categoriesService = categoriesService;

    }
    @GetMapping(path = "/getAllCategoriesOfBooks")
    public ResponseEntity<List<Category>> getAllCategories(){
        List<Category> categories = categoriesService.getAllCategoriesOfBooks();
return new ResponseEntity<>(categories, HttpStatus.OK);     }

    @GetMapping(path = "/getAllBooksByCategories/{categoryId}")     public
ResponseEntity<List<Books>> getAllBooksByCategories(@PathVariable Long
categoryId){
```

```
        List<Books> books = categoriesService.getAllBooksByCategories(categoryId);
return new ResponseEntity<>(books,HttpStatus.OK);     }

    @PostMapping(path = "/addBooks")     public ResponseEntity<Books>
addBooks(@RequestBody AddBooksDto addBooksDto){         return new
ResponseEntity<>(studentService.addBooks(addBooksDto),HttpStatus.OK);
    }

    @DeleteMapping(path = "/removeBooksFromStudent/{bookId}")     public
ResponseEntity<Books> removeBooksFromStudent( @PathVariable Long bookId){
        return new
ResponseEntity<>(this.studentService.removeBooksFromStudent(bookId),HttpStatus.OK);
    }

    @GetMapping(path = "/getStudentDetails/{studentId}")     public
ResponseEntity<StudentDetailsDto> getStudentDetails(@PathVariable Long
studentId){
        StudentDetailsDto studentDetailsDto =
studentService.getStudentDetails(studentId);         return new
ResponseEntity<>(studentDetailsDto,HttpStatus.OK);
    }
}
```

3.2 Testing of API's

Testing is a crucial aspect of the development process, ensuring that the APIs function as expected and handle edge cases gracefully. The tests implemented for the authentication and student controllers include unit tests and integration tests.

AuthenticationControllerTest.java

```
 @Test
    void registerStudent() {
        RegisterRequest registerRequest = new RegisterRequest("John Doe", "12345",
"john@example.com", "password");
         when(authenticationService.registerStudent(registerRequest)).thenReturn("St
udent registered successfully");
```

```
        ResponseEntity<String> responseEntity =
authenticationController.registerStudent(registerRequest);
         assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
assertEquals("Student registered successfully", responseEntity.getBody());
    }
```

```
@Test
    void loginStudent() {
        LoginRequest loginRequest = new LoginRequest("john@example.com",
"password");
        LoginResponse loginResponse = new LoginResponse("Login Successful", 1L);

when(authenticationService.loginStudent(any(LoginRequest.class))).thenRetur
n(loginResponse);

        ResponseEntity<LoginResponse> responseEntity =
authenticationController.loginStudent(loginRequest);
         assertEquals(HttpStatus.OK,
responseEntity.getStatusCode());
assertEquals(loginResponse, responseEntity.getBody());     }
```

StudentControllerTest.java

```
@Test
    void getAllCategories() {
        List<Category> categories = new ArrayList<>();
        // Add some categories to the list

        when(categoriesService.getAllCategoriesOfBooks()).thenReturn(categories);
        ResponseEntity<List<Category>> responseEntity =
studentController.getAllCategories();

        assertEquals(HttpStatus.OK,
responseEntity.getStatusCode());         assertEquals(categories,
responseEntity.getBody());     }
```

1.getAllCategories():

Tests the retrieval of all book categories.
Asserts that the HTTP status code is 200 OK and the response body matches the expected list of categories.

2.getAllBooksByCategories():

Tests the retrieval of all books under a specific category.
Asserts that the HTTP status code is 200 OK and the response body matches the expected list of books.

3.addBooks():

Tests the addition of new books.
Asserts that the HTTP status code is 200 OK.

3.3 Backend pipeline script

```
pipeline {
agent any
environment {
        DOCKER_HUB_USERNAME = 'tezaswa'
        KUBECONFIG="/home/tezaswa/.kube/config"
    }     tools {
nodejs 'node16'
jdk 'jdk17'
    }
    stages {         stage('Clean
Workspace') {              steps {
cleanWs()
            }
        }
        stage('Pull Git Repo') {              steps {                 git
'https://github.com/Tezaswa06/Devops_Major_Project.git'
            }
        }
        stage('Maven Build and Test') {
steps {                 dir('Backend') {
script {                        sh 'mvn
clean install'                          sh 'mvn
test'
                }
            }
        }
    }
        stage('Docker Build and Run using docker-compose') {
steps {                 script {                    sh 'docker-compose
pull'                   sh 'docker-compose build'
            }
```

```
            }
        }
        stage('Trivy Scan') {
steps {                 script {
try {
                    sh 'trivy fs --severity HIGH,CRITICAL .'
                } catch (Exception e) {
echo "Trivy scan failed: ${e.message}"
                }
            }
        }
    }
    stage('Pull Docker Image of Client using Ansible') {
steps {
            ansiblePlaybook becomeUser: null, colorized: true, disableHostKeyChecking: true, inventory: './inventory',
playbook: './clientPlaybook.yml', sudoUser: null, vaultTmpPath: ''
        }
    }
    stage('Pull Docker Image of Server using Ansible') {
steps {
            ansiblePlaybook becomeUser: null, colorized: true, disableHostKeyChecking: true, inventory: './inventory',
playbook: './serverPlaybook.yml', sudoUser: null, vaultTmpPath: ''
        }
    }
  }
}
```

4 Frontend

4.1 Dependecies Overview

Frontend development involves setting up the project configuration, including package management and build tools.

Package.json

The package.json file manages project dependencies and scripts for building and testing the frontend application.

```
{
  "name": "frontend",
"private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "test": "jest",
    "test:books": "jest --env=jsdom src/Books.test.jsx",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --
maxwarnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "antd": "^5.17.3",
    "axios": "^1.7.2",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.23.1"
  },
  "devDependencies": {
    "@babel/preset-env": "^7.24.5",
    "@babel/preset-react": "^7.24.1",
    "@testing-library/jest-dom": "^6.4.5",
    "@testing-library/react": "^15.0.7",
    "@types/react": "^18.2.66",
    "@types/react-dom": "^18.2.22",
    "@vitejs/plugin-react": "^4.2.1",
    "autoprefixer": "^10.4.19",
    "axios-mock-adapter": "^1.22.0",
    "babel-jest": "^29.7.0",
    "eslint": "^8.57.0",
    "eslint-plugin-react": "^7.34.1",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.6",
    "jest": "^29.7.0",
    "jest-environment-jsdom": "^29.7.0",
    "postcss": "^8.4.38",
    "tailwindcss": "^3.4.3",
    "vite": "^5.2.0"
  },
  "jest": {
    "transform": {
```

```
      "^.+\\.jsx?$": "babel-jest"
    }
  }
}
```

4.2 Config Files

Babel.rc
Babel.rc is a configuration file for Babel, a JavaScript compiler. This section explains the Babel configuration used in the project.

```
{
    "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

4.3 Frontend Testing

Frontend testing is a crucial aspect of the software development lifecycle, ensuring that the user interface (UI) behaves as expected and provides a seamless experience to users. Effective frontend testing helps identify and rectify bugs early in the development process, improving the overall quality and reliability of the application.

Books.tests.jsx Books.tests.jsx handles the configuration of various components within the frontend application.

```
import React from 'react'; import { render }
from '@testing-library/react'; import Books
from './Components/Books';


describe('Books Component', () => {
it('renders without crashing', () => {
render(<Books />);
  }); });
```

Jestconfig.jsx

Jestconfig.jsx is used for configuring Jest, a testing framework for JavaScript. This section outlines the Jest configuration for the project.

```
module.exports = {
testEnvironment: 'jsdom',
transform: {
      "^.+\\.jsx?$": "babel-jest"
    },
    moduleFileExtensions: ["js", "jsx"],
testMatch: ["*/src/.test.js", "*/src/.test.jsx"]
  };
```

4.4 Frontend pipeline script

The frontend pipeline script automates the build and deployment of the frontend application, ensuring a seamless integration with the backend services.

```
pipeline {
agent any
environment {
      DOCKER_HUB_USERNAME = 'tezaswa'
   }    tools {
nodejs 'node16'
jdk 'jdk17'
   }
   stages {       stage('Clean
Workspace') {           steps {
cleanWs()
        }
      }
      stage('Pull Git Repo') {          steps {              git
'https://github.com/Tezaswa06/Devops_Major_Project.git'
        }
      }
      stage('Maven Build and Test') {
steps {              dir('Backend') {
script {                    sh 'mvn
clean install'                    sh 'mvn
test'
            }
          }
```

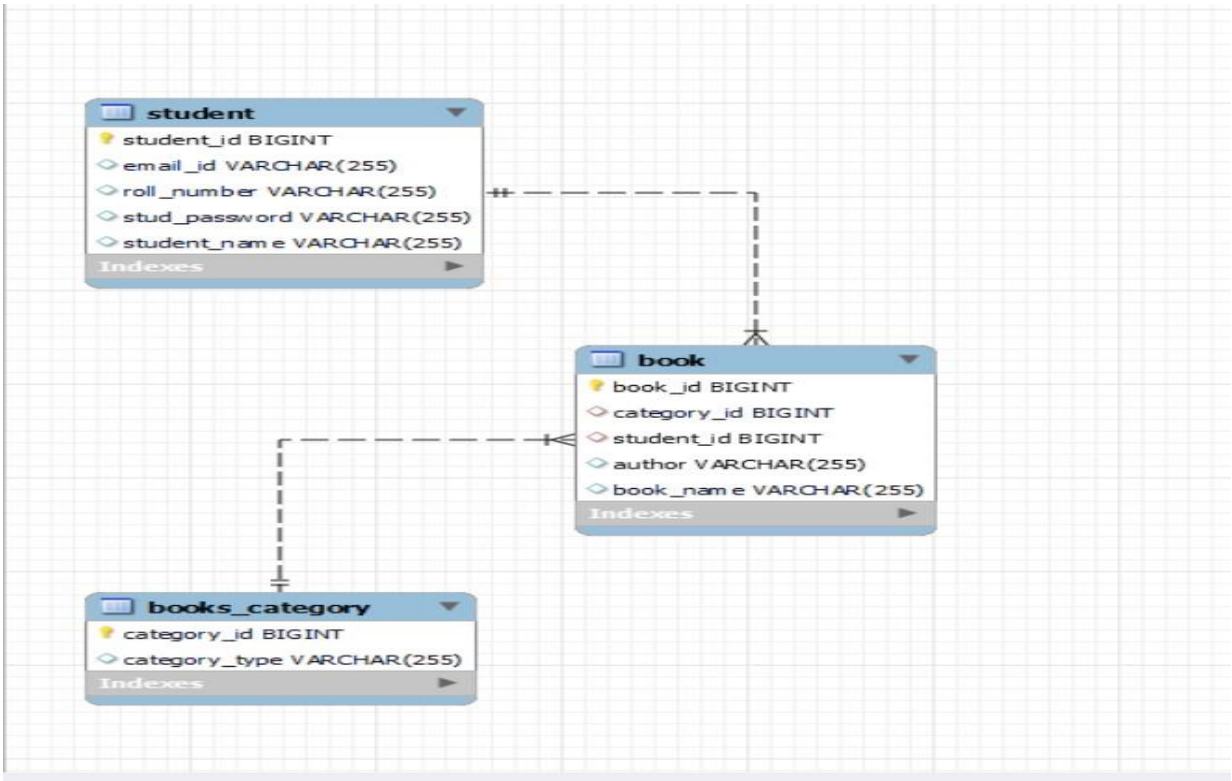
```
        }
    }
    stage('Dependency Installation for Frontend') {
        steps {
dir('Frontend') {
            script {
                sh 'npm install --save'
sh 'npm install --save-dev jest-environment-jsdom --force'
sh 'npm install --save-dev jest@latest ts-jest@latest --force'
sh 'npm install -g ts-jest --force'
            }
        }
    }
}
    stage('Frontend Testing') {
steps {
        dir('Frontend') {
script {
            sh 'npm run
test:books'
            }
        }
    }
}
    stage('Docker Build and Run using Docker Compose')
{
    steps {
        script {
            sh 'docker-compose pull'
sh 'docker-compose build'
        }
    }
}
    stage('Trivy Scan') {
steps {
        script {
try {
            sh 'trivy fs --severity HIGH,CRITICAL .'
        } catch (Exception e) {
echo "Trivy scan failed: ${e.message}"
        }
    }
}
}
    stage('Tag and Push Docker Images') {
steps {
        script {
        sh "docker tag library_managment_system_backend
${DOCKER_HUB_USERNAME}/library_managment_system_backend"
        withCredentials([usernamePassword(credentialsId: 'docker-jenkins', usernameVariable: 'DOCKER_HUB_USERNAME',
passwordVariable: 'DOCKER_HUB_PASSWORD')]) {
            sh "docker login -u '${DOCKER_HUB_USERNAME}' -p '${DOCKER_HUB_PASSWORD}'"
sh "docker push ${DOCKER_HUB_USERNAME}/library_managment_system_backend"
        }


        sh "docker tag library_managment_system_frontend
${DOCKER_HUB_USERNAME}/library_managment_system_frontend"
        withCredentials([usernamePassword(credentialsId: 'docker-jenkins', usernameVariable: 'DOCKER_HUB_USERNAME',
passwordVariable: 'DOCKER_HUB_PASSWORD')]) {
            sh "docker login -u '${DOCKER_HUB_USERNAME}' -p '${DOCKER_HUB_PASSWORD}'"
sh "docker push ${DOCKER_HUB_USERNAME}/library_managment_system_frontend"
        }


        sh "docker tag mysql ${DOCKER_HUB_USERNAME}/mysql"
        withCredentials([usernamePassword(credentialsId: 'docker-jenkins', usernameVariable: 'DOCKER_HUB_USERNAME',
passwordVariable: 'DOCKER_HUB_PASSWORD')]) {
```

```
                    sh "docker login -u '${DOCKER_HUB_USERNAME}' -p '${DOCKER_HUB_PASSWORD}'"
sh "docker push ${DOCKER_HUB_USERNAME}/mysql"
                }
            }
        }
    }
    stage('List Docker Images') {
steps {                script {
sh 'docker images'
            }
        }
    }
    stage('Pull Docker Image of Nodes using Ansible') {
steps {
            ansiblePlaybook becomeUser: null, colorized: true, disableHostKeyChecking: true, inventory: './inventory',
playbook: './docker-deploy.yml', sudoUser: null, vaultTmpPath: ''
        }
    }
}
}
```

5 Database

5.1 Database Design

Database design involves structuring the database to efficiently store and retrieve data. This section outlines the database schema, tables, and relationships used in the project.



- The student table stores information about students.
- The book table stores information about books and their associations with students and categories.
- The books_category table stores information about different categories of books.
- Relationships between tables are maintained through foreign keys (student_id and category_id) in the book table, linking it to the student and books_category tables respectively.

6 Deployment Automation

Deploy automation streamlines software delivery by leveraging tools like Docker, Kubernetes, Ansible, Grafana, and Jenkins. Docker enables the packaging of applications into portable containers, while Kubernetes orchestrates these containers, ensuring scalable and reliable deployments. Ansible automates the configuration and provisioning of infrastructure, ensuring consistency across environments. Grafana provides monitoring and visualization, allowing teams to track application performance and set up alerts for anomalies. Jenkins facilitates continuous integration and continuous delivery (CI/CD), automating the build, test, and deployment processes. Together, these tools create a robust pipeline that enhances efficiency, reliability, and speed in software delivery, ensuring that applications are consistently built, tested, deployed, and monitored across various environments.

7 Jenkins Pipeline

The Jenkins pipeline displayed in the screenshot represents a series of automated stages for building, testing, and deploying a project. Here's a brief explanation of each stage in the pipeline:
Declarative: Tool Install
Purpose: This stage installs any necessary tools required for the build process.
Execution Time: Approximately 272ms - 365ms.
Clean Workspace
Purpose: This stage ensures that the workspace is cleaned, removing any previous build artifacts or unnecessary files to ensure a fresh start.
Execution Time: Approximately 424ms - 559ms.
Pull Git Repo
Purpose: This stage pulls the latest code from the Git repository.
Execution Time: Approximately 9s - 31s.
Maven Build and Test

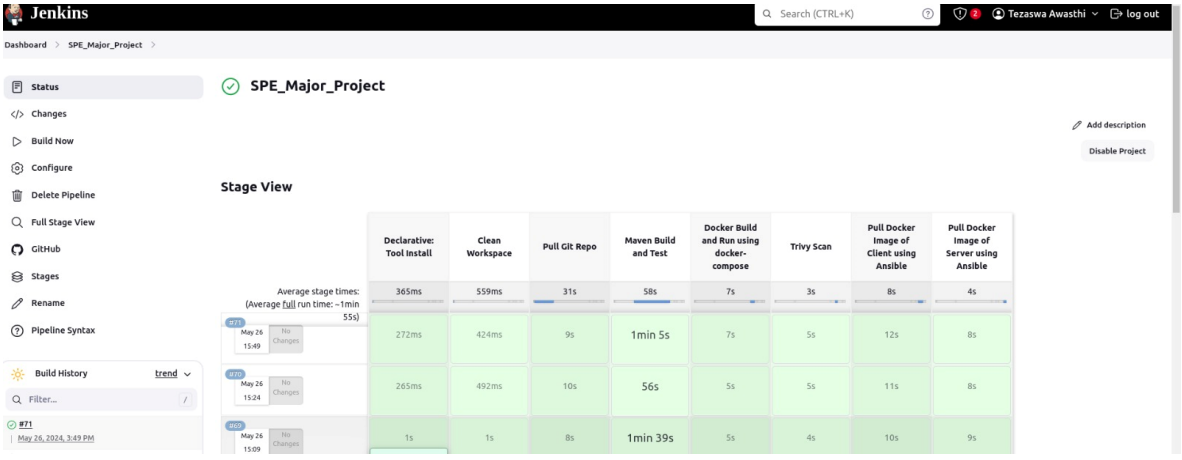
Purpose: This stage compiles the code and runs tests using Maven, a build automation tool.Execution Time: Approximately 56s - 1min 5s.
Docker Build and Run using docker-compose

Purpose: This stage builds Docker images and starts containers using docker-compose, a tool for defining and running multi-container Docker applications.
Execution Time: Approximately 7s.
Trivy Scan

Purpose: This stage runs Trivy, a vulnerability scanner for Docker images, to check for security issues.
Execution Time: Approximately 3s - 5s.
Pull Docker Image of Client using Ansible

Purpose: This stage pulls the Docker image for the client component using Ansible, an automation tool.
Execution Time: Approximately 8s - 12s.
Pull Docker Image of Server using Ansible

Purpose: This stage pulls the Docker image for the server component using Ansible.
Execution Time: Approximately 4s - 10s.
Key Observations:
The pipeline ensures that each step in the build, test, and deployment process is automated and repeatable.
The average full run time is around 1 minute 55 seconds, indicating a fairly quick and efficient pipeline.
The stages are designed to handle different aspects of the software development lifecycle, from code fetching and building to security scanning and deployment.
This setup is typical for Continuous Integration (CI) and Continuous Deployment (CD) workflows, where code changes are automatically tested and deployed to ensure quick feedback and delivery.



8 Docker

Docker Installation

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-
buildx-plugin docker-compose-plugin
```

Docker Files

Docker files are essential for defining the environment in which the application runs. These files contain instructions on how to set up the application within a Docker container, including the base image, dependencies, and configuration settings.

Creating Docker files for each service in the applicarion

Docker file for backend

```
FROM openjdk:17-jdk-alpine
WORKDIR /app
COPY ./target/Library-Managment-System-0.0.1-SNAPSHOT.jar ./app.jar
EXPOSE 8081 ENTRYPOINT ["java","-jar","app.jar"]
```

Dockerfile for frontend

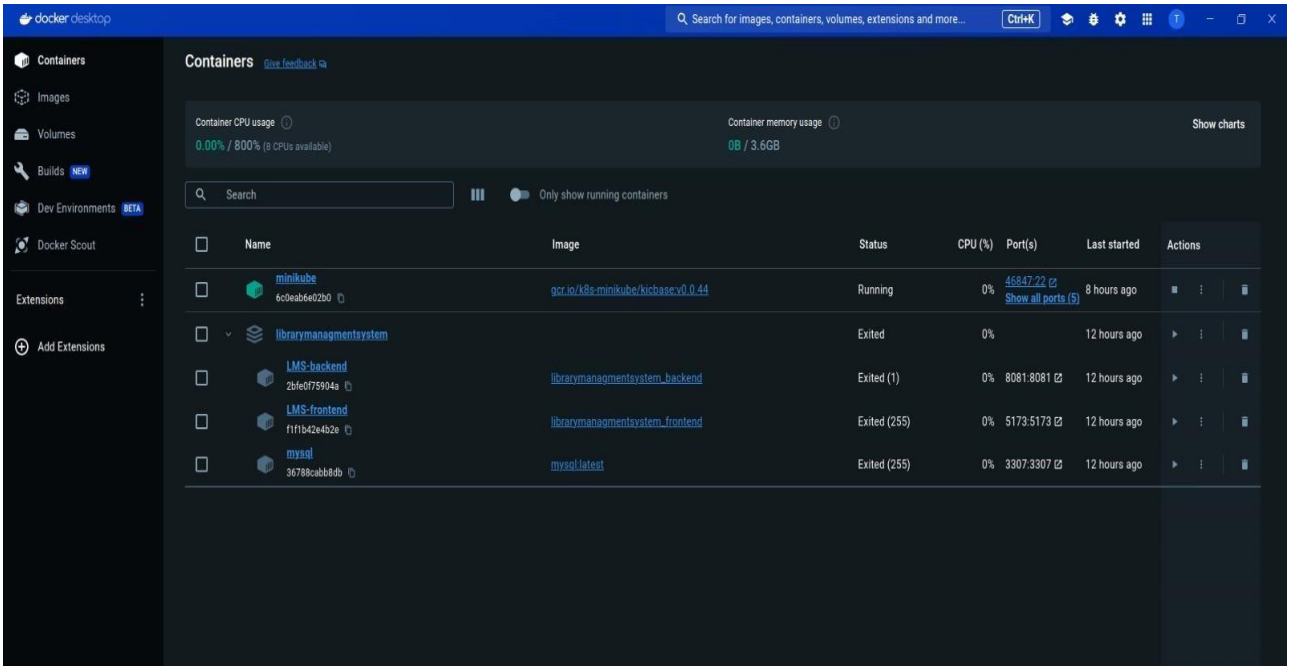
```
# Use an official Node.js runtime as the base image
FROM node:18-alpine

# Set the working directory in the container
WORKDIR /app

# Copy package.json and package-lock.json to the container
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code to the container COPY . .

# Build the React app
RUN npm run build

# Expose the port on which the React app will run
EXPOSE 5173

# Command to run the React app
```

Docker Pipeline Script

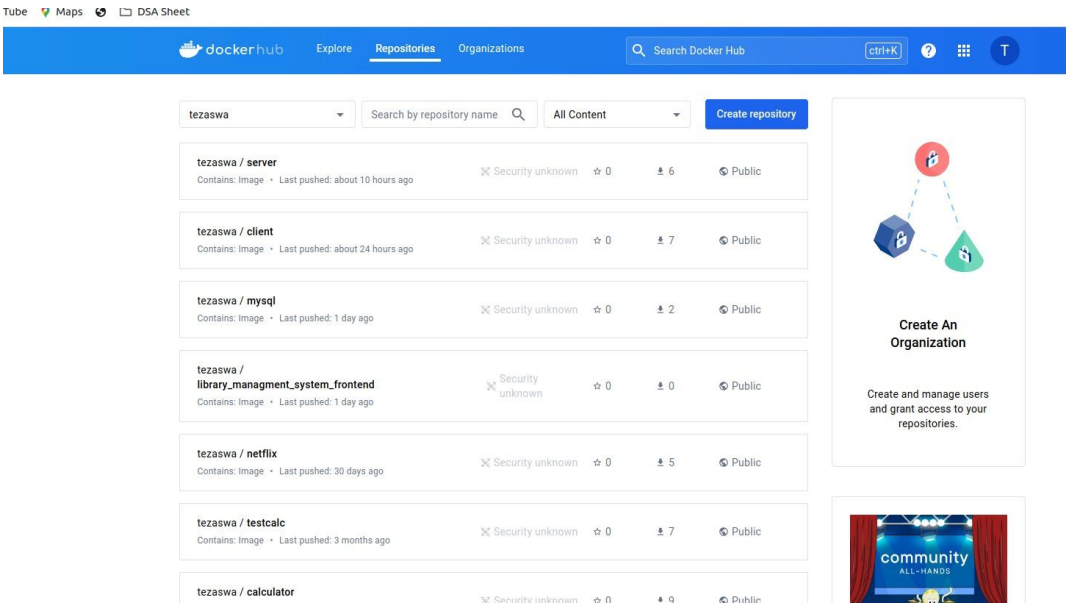
The Docker pipeline script automates the building, testing, and deployment of Docker images. This script ensures that the application is consistently packaged and deployed across different environments.

```
stage('Docker Build and Run using docker compose') {
            steps {
                script {
                    sh 'docker-compose pull'
                    sh 'docker-compose build'
                }
            }
        }
        stage('Trivy Scan') {
            steps {
                script {
                    try {
                        sh 'trivy fs --severity HIGH,CRITICAL .'
                    } catch (Exception e) {
                        echo "Trivy scan failed: ${e.message}"
                    }
                }
            }
        }
        stage('Tag and Push Docker Images') {
            steps {
                script {
                    sh "docker tag library_managment_system_backend ${DOCKER_HUB_USERNAME}/library_managment_system_backend"
                    withCredentials([usernamePassword(credentialsId: 'docker-jenkins', usernameVariable: 'tezaswa', passwordVariable: '915368742@Awasthi')]) {
                        sh "docker login -u 'tezaswa' -p '915368742@Awasthi' "
                        sh "docker push ${DOCKER_HUB_USERNAME}/library_managment_system_backend"
                    }

                    // Tag and push the frontend image
                   sh "docker tag library_managment_system_frontend ${DOCKER_HUB_USERNAME}/library_managment_system_frontend"
                    withCredentials([usernamePassword(credentialsId: 'docker-jenkins', usernameVariable: 'tezaswa', passwordVariable: '915368742@Awasthi')]) {
                        sh "docker login -u 'tezaswa' -p '915368742@Awasthi'"
                        sh "docker push ${DOCKER_HUB_USERNAME}/library_managment_system_frontend"
                    }

                    // Tag and push the mysql image
                   sh "docker tag mysql ${DOCKER_HUB_USERNAME}/mysql"
                    withCredentials([usernamePassword(credentialsId: 'docker-jenkins', usernameVariable: 'tezaswa', passwordVariable: '915368742@Awasthi')]) {
                        sh "docker login -u 'tezaswa' -p '915368742@Awasthi'"
                        sh "docker push ${DOCKER_HUB_USERNAME}/mysql"
                    }
                }
            }
        }
        stage('List Docker Images') {
            steps {
                script {
```

Containers and Images



Images



9 Ansible

Ansible is a powerful automation tool used to streamline the process of application deployment, configuration management, and orchestration. In our project, Ansible plays a crucial role in ensuring that our DevOps practices are efficient, reliable, and scalable. Here's an overview of how Ansible is utilized in our project:

Automation of Deployment Processes
Ansible simplifies the deployment process by automating the steps required to deploy our application components. This includes setting up the environment, pulling the latest Docker images, and ensuring that all services are correctly configured and running. By using Ansible playbooks, we can define these processes in a simple, readable format that is easy to maintain and share across the team.

Configuration Management

One of the key advantages of Ansible is its ability to manage configuration files and settings across multiple environments consistently. In our project, Ansible ensures that all nodes in our infrastructure are configured uniformly, reducing the risk of configuration drift and improving overall system reliability. This is particularly important for maintaining consistency between development, testing, and production environments.

Orchestration of Multi-Container Applications
Our application consists of multiple services that need to be orchestrated to work together seamlessly. Ansible helps orchestrate these services, ensuring that each container is deployed in the correct order and that dependencies are properly managed. This orchestration capability is vital for maintaining the integrity of our application's microservices architecture.

Integration with CI/CD Pipeline
Ansible is integrated into our CI/CD pipeline to automate the deployment of both the client and server components of our application. This integration ensures that new code changes are automatically deployed to the appropriate environments after passing the necessary tests. By incorporating Ansible into our pipeline, we reduce manual intervention, accelerate deployment times, and increase the overall reliability of our deployments.

Clientplaybook.yml

This playbook pulls the latest Docker image for the client application, configures the necessary environment variables, and starts the client container.

```
---
-     name: Deploy React Application   hosts:
localhost    vars:
    docker_image:
"tezaswa/client:latest"
container_name: "client"   tasks:

-     name: Stop and remove existing
container       docker_container:
name: "{{ container_name }}"         state:
absent         force_kill: yes

-     name: Run Docker container
docker_container:
        name: "{{ container_name }}"
image: "{{ docker_image }}"


        state: started         restart_policy: always         pull: true
ports:
          - "5173:5173"
```

Serverplaybook.yml

Similar to the client playbook, this playbook pulls the latest Docker image for the server, configures the environment, and starts the server container.

```
---
-     name: Deploy Spring Boot Application   hosts:
localhost    vars:
    docker_image: "tezaswa/server:latest"
container_name: "server"   tasks:

-     name: Stop and remove existing container
docker_container:         name: "{{ container_name
}}"         state: absent         force_kill: yes

-     name: Run Docker container
docker_container:
        name: "{{ container_name }}"
image: "{{ docker_image }}"
state: started         restart_policy:
always         pull: true         ports:
-     "8081:8081"

 apiVersion: v1 kind:
Service metadata:
  name: server-service   namespace:
library-management spec:   selector:
    app: server    ports:
    - protocol: TCP
port: 8888       targetPort:
8081   type: LoadBalancer
```

10 Kubernetes

Setting Up a Kubernetes Cluster for the Backend Server
Deploying our backend server as a Kubernetes cluster ensures scalability, high availability, and efficient management of resources. Here's a comprehensive guide to setting up a Kubernetes cluster for our backend server.

Prerequisites
Kubernetes Installed: Ensure that Kubernetes is installed on your local machine or cloud environment. This can be done using tools like Minikube for local development or managed Kubernetes services such as Google Kubernetes Engine (GKE), Amazon EKS, or Azure AKS.
Kubectl Installed: Install kubectl, the Kubernetes command-line tool, to interact with the cluster.
Docker Installed: Docker must be installed to build and push images to a container registry.
Step-by-Step Setup
Create a Kubernetes Cluster

For local development, you can use Minikube:
$minikube start
For a managed service, follow the provider-specific instructions to create a cluster. Configure kubectl

Ensure that kubectl is configured to interact with your cluster:

$kubectl config use-context <your-cluster-context> Build
and Push Docker Images

Navigate to the backend project directory and build the Docker image:
$docker build -t tezaswa/backend:latest .
Push the Docker image to a container registry (e.g., Docker Hub, GCR, ECR):
$docker push tezaswa/backend:latest

Kubernetes.yml

```
---
- name: Deploy Spring Boot Application to Kubernetes   hosts:
  localhost   collections:     - kubernetes.core
  environment:
    KUBECONFIG: "/home/tezaswa/.kube/config"   tasks:
- name: Delete existing Deployment if present
  kubernetes.core.k8s:
        state: absent         kind:
Deployment         name: server
namespace: library-management
ignore_errors: yes

-     name: Delete existing Service if present
kubernetes.core.k8s:
        state: absent         kind:
Service         name: server-service
namespace: library-management
ignore_errors: yes

-     name: Apply Deployment
kubernetes.core.k8s:
        state: present         definition: "{{ lookup('file',
'serverDeployment.yml') }}"
-     name: Apply Service
kubernetes.core.k8s:
        state: present         definition: "{{ lookup('file',
'serverService.yml') }}"
```

serverDeployment.yml

```
apiVersion: apps/v1 kind:
Deployment metadata:
  name: server   namespace:
library-management spec:
  replicas: 3
selector:
matchLabels:
app: server
template:
metadata:



      labels:
        app: server
spec:
containers:
-     name: server-kubernetes         image:
tezaswa/server:latest         ports:
-     containerPort: 8081
```

severService.yml

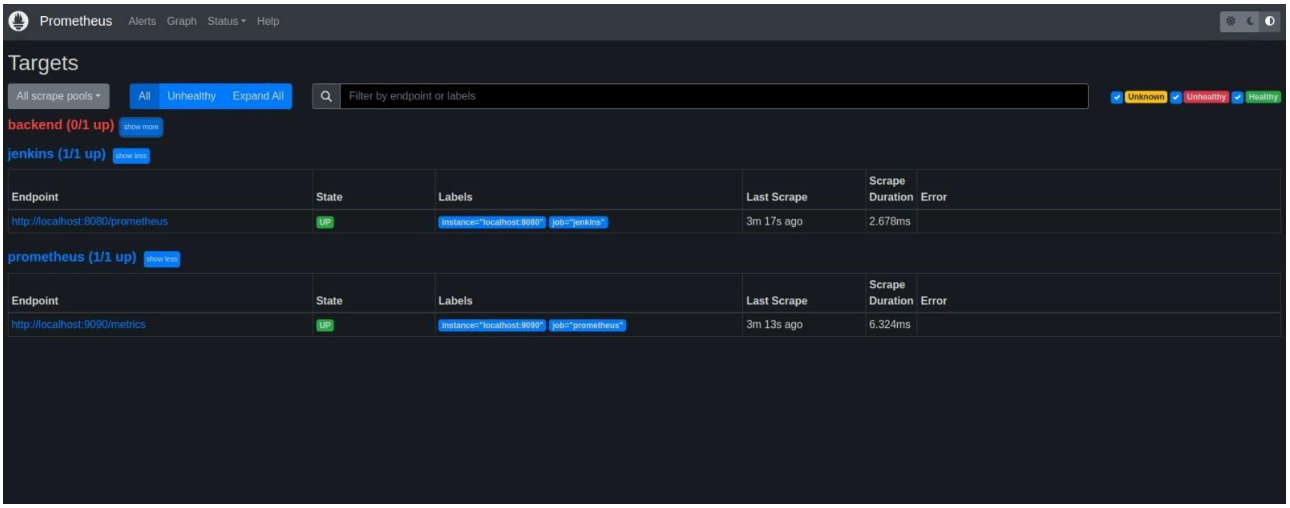
```
apiVersion: v1 kind:
Service metadata:
  name: server-service
namespace: library-management
spec:   selector:
    app: server
ports:
    - protocol: TCP
port: 8888
targetPort: 8081   type:
LoadBalancer
```

serverplaybook.yml

```
---
-     name: Deploy Spring Boot
Application   hosts: localhost    vars:
    docker_image:
"tezaswa/server:latest"
container_name: "server"   tasks:

-     name: Stop and remove existing
container       docker_container:
name: "{{ container_name }}"
state: absent

        force_kill: yes

- name: Run Docker container       docker_container:
        name: "{{ container_name }}"         image: "{{ docker_image }}"
state: started         restart_policy: always         pull: true         ports:
- "8081:8081"
```

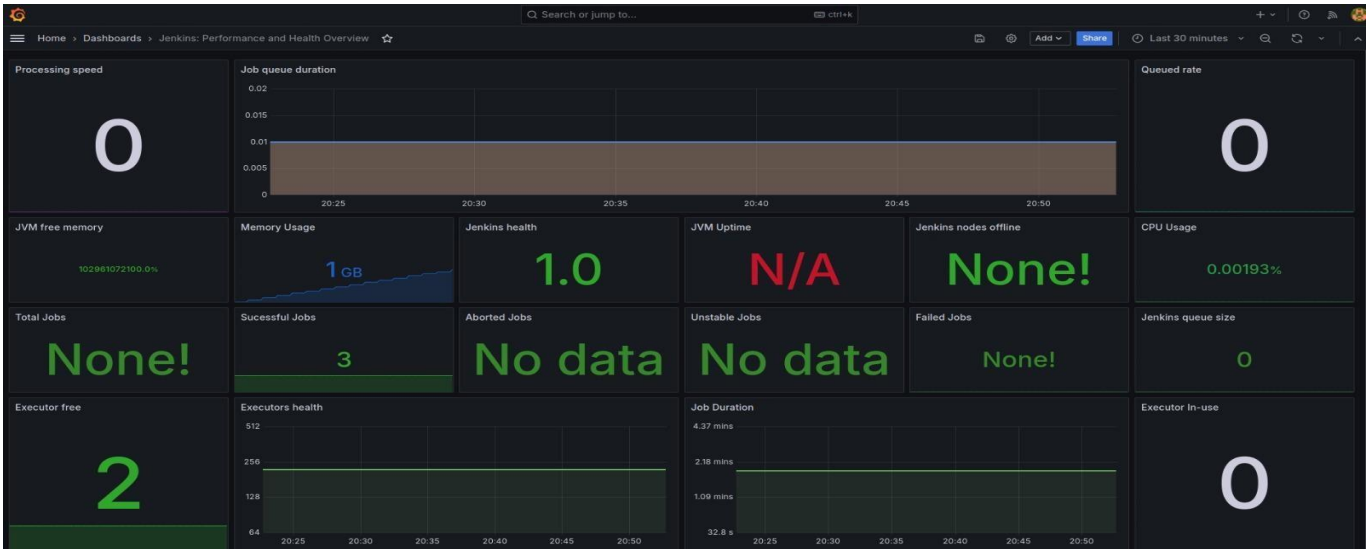
11 Moneitoring

1. Prometheus



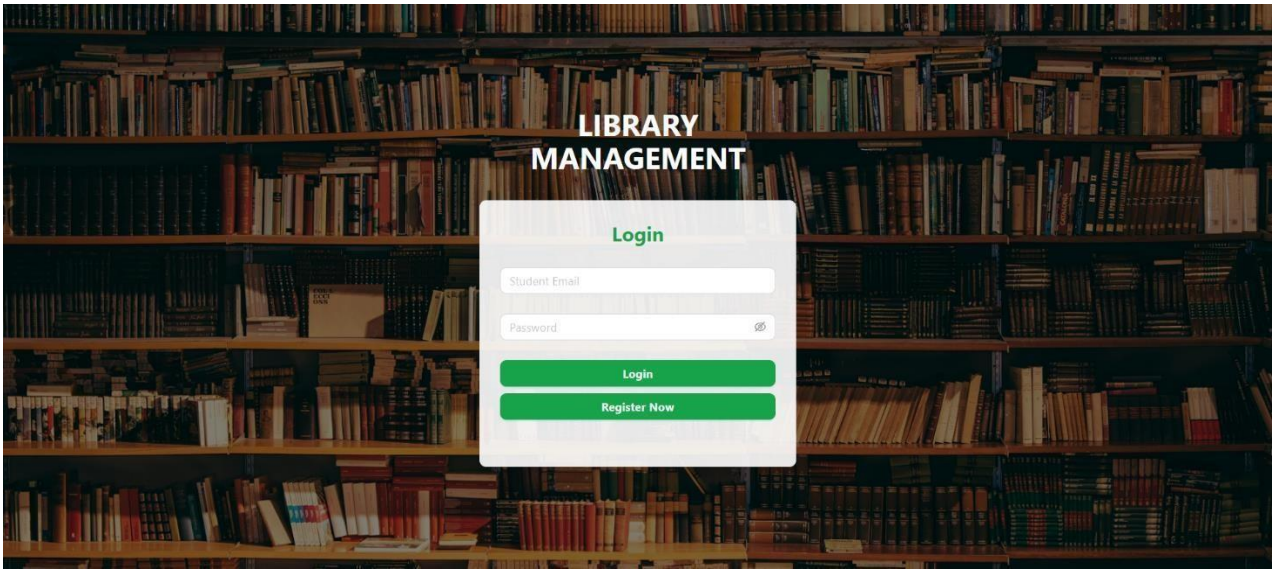
The Jenkins and Prometheus targets are up and being scraped successfully, while the backend service is down, indicating a possible issue with that service. The labels help in identifying and grouping the targets, and the scrape details provide insight into the performance and status of the scraping process.

Graphana

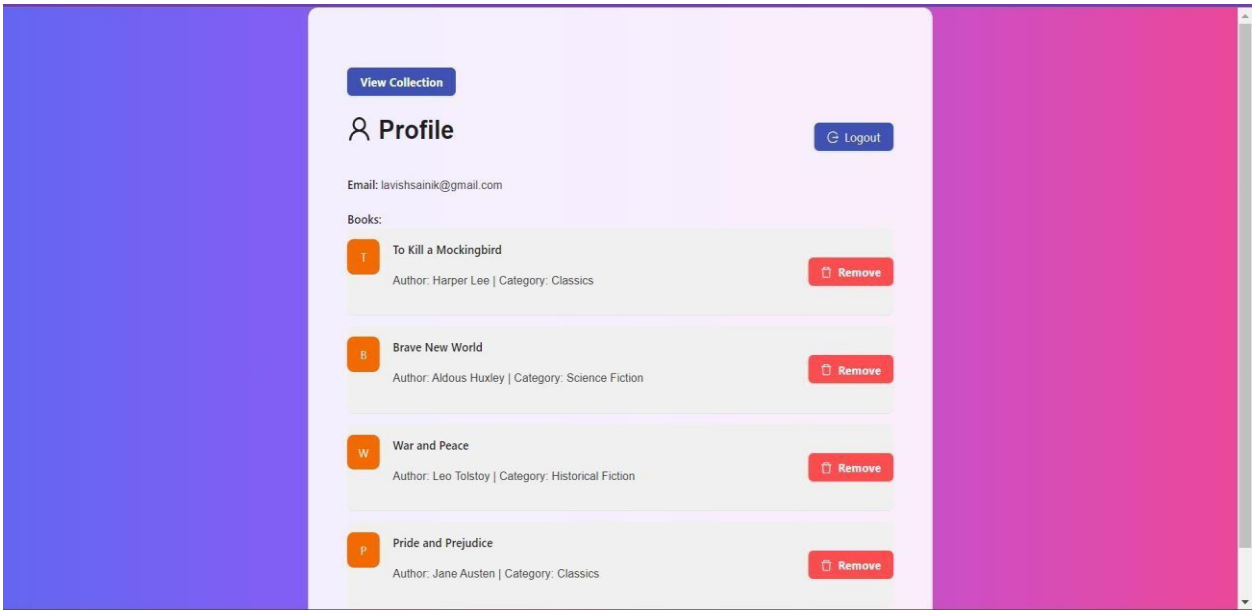


12 Results and discussion

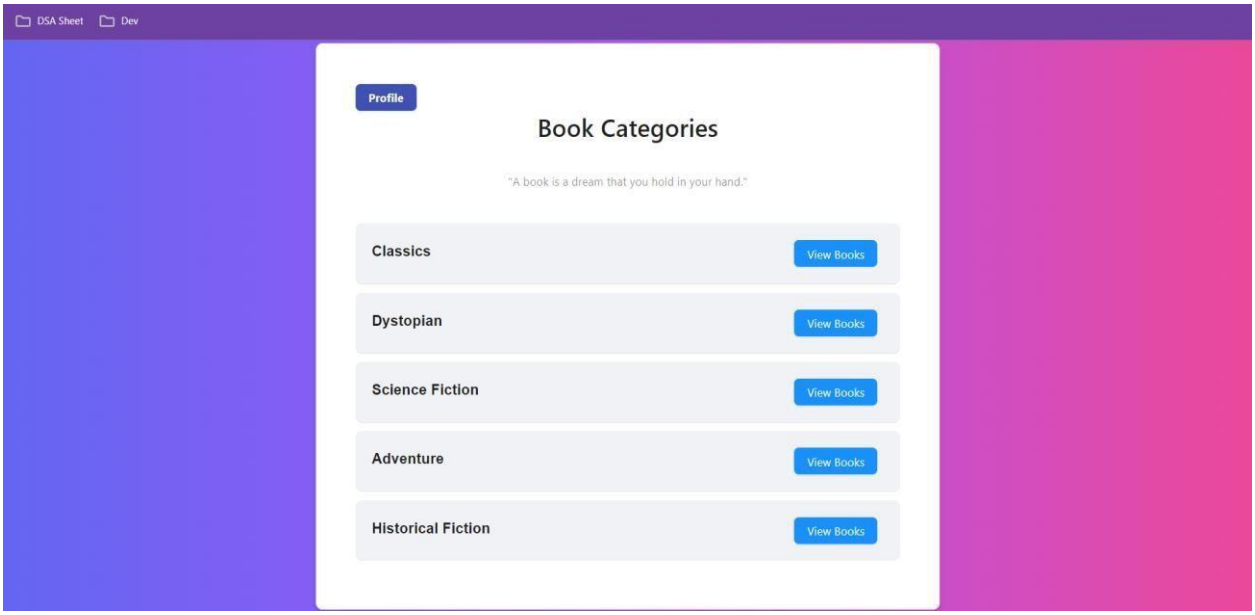
11.1 Login Page



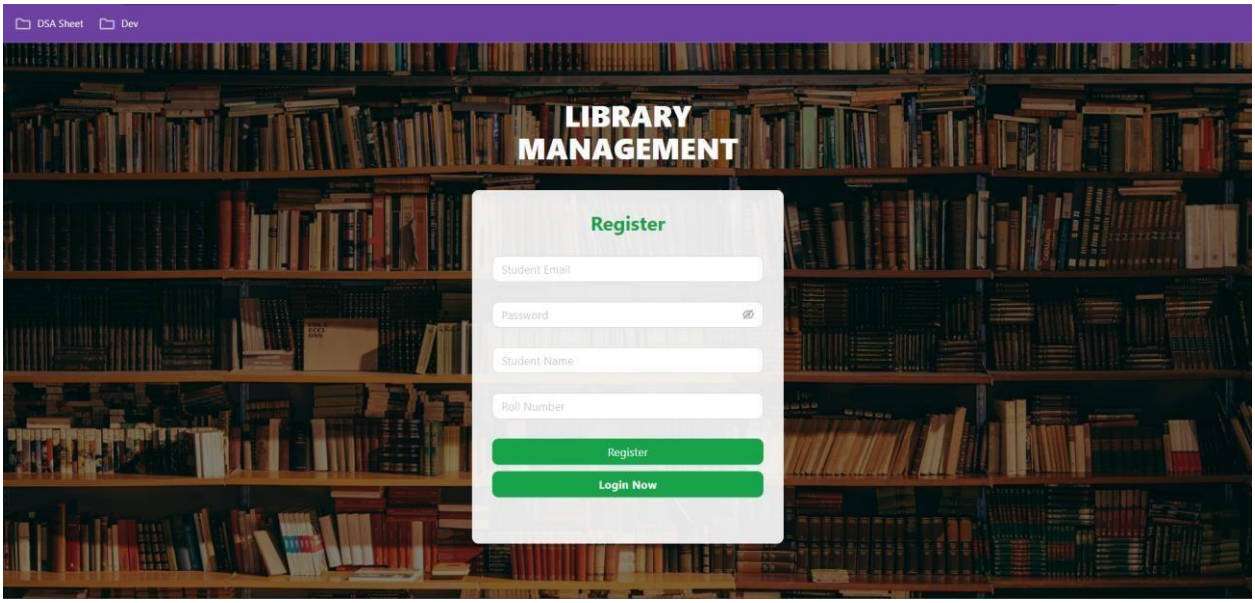
11.2 Profile



11.3 Collection



11.4 Register



12 References