Software Production Engineering Final Project Report

Munagala Kalyan Ram, IMT2021023
Vikas Kalyanapuram, IMT2021040

December 17, 2024

1 Links and References

1. **GitHub Repo**: https://github.com/KalyanRam1234/SPE_Final_Project
2. **Dockerhub**:
   (a) https://hub.docker.com/u/vikaskaly
   (b) https://hub.docker.com/u/kalyanramm237

2 Introduction

2.1 Description of the Application

The Library Management Service is an application designed to streamline the process of borrowing and managing books within a library. It is built using a microservice architecture, with separate services for handling book details, student information, and book lending operations.

The backend services are developed in Java (with unit tests in JUnit) and utilize Spring Boot for creating RESTful APIs, while the frontend is built with React. The application uses MySQL for data storage and is containerized using Docker to ensure consistency across different environments.

2.2 Objective

1. **Containerization**: Use Docker to containerize the application for consistency across different environments.
2. **Orchestration**: Employ Kubernetes to manage the deployment, scaling, and operation of the microservices.
3. **Continuous Integration**: Set up a CI pipeline using Jenkins to automate the build and testing process.

4. **Continuous Deployment**: Implement a CD pipeline using Ansible to automate the deployment process.

5. **Logging and Error Handling**: Integrate robust logging and error handling mechanisms to maintain application reliability.

3 Project Functionality

3.1 Book-Application

3.1.1 Functionality

- The API endpoints allow users to add, update, and delete books.
- Each book has the following attributes:
  - Book Code (unique for each book)
  - Book Title
  - Book Author
  - Book Description

3.1.2 Endpoints

The following are the API endpoints in our application:

Method	Endpoint	Description
GET	/api/books	Returns all books that have been added to the database so far.
GET	/api/books/code/`code`	Returns the book that corresponds to the code in the path variable. Returns a BookNotFoundException if no book exists.
POST	/api/books	Takes a book object in the body of the request and adds it to the database. Returns a BookAlreadyExistsException if a book with the same code exists.
PUT	/api/books	Takes a book object and updates the corresponding book in the database. Returns a BookNotFoundException if the book is not found.
DELETE	/api/books/code/`code`	Deletes the book that corresponds to the code in the path variable. Returns a BookNotFoundException if the book is not found.

3.1.3 Project Structure

- Follows the Controller-Service-Repository pattern:
  - The **controller layer** defines the REST API endpoints and manages the interactions with the service layer.
  - The **service layer** handles the business logic and follows SOLID principles. It includes the Book Service Implementation and logs different operations using slf4j.
  - The **repository layer** handles interactions with the database. The JPA library is used to facilitate database operations.

3.1.4 Testing

Testing is performed as follows (using JUnit to automate the tests):

- **Unit Tests**: Test methods in the repository layer using an in-memory H2 database to verify CRUD operations.
- **Service Layer Tests**: Mock the repository layer (using Mockito) to isolate the business logic and validate business rules.
- **Integration Tests**: Test the API endpoints (using MockMVC) to verify correct handling of requests and responses. Ensure correct implementation of all HTTP methods (GET, POST, PUT, DELETE).

3.2 Student-Application

3.2.1 Functionality

The API endpoints allow users to add, update, and delete students. Each student has the following attributes:

- Student RollNo (unique for each student)
- Student Name
- Student Email
- Student Phone

3.2.2 Endpoints

The following are the API endpoints in our application:

Method	Endpoint	Description
GET	/api/students	Returns all students that have been added to the database so far.

Method	Endpoint	Description
GET	/api/students/rollNo/`rollNo`	Returns the student that corresponds to the rollNo in the path variable. Returns a StudentNotFoundException if no student exists.
POST	/api/students	Takes a student object in the body of the request and adds it to the database. Returns a StudentAlreadyExistsException if a student with the same rollNo exists. Returns a NullFieldsException if any attribute of the student is null in the request.
PUT	/api/students	Takes a student object and updates the corresponding student in the database. Returns a StudentNotFoundException if no student is found. Returns a NullFieldsException if any attribute of the student is null in the request.
DELETE	/api/students/rollNo/`rollNo`	Deletes the student that corresponds to the rollNo in the path variable from the database. Returns a StudentNotFoundException if no student is found.

3.2.3 Project Structure

The project follows the Controller-Service-Repository pattern:

- **Controller Layer:** Manages the REST interface to the business logic. Defines the REST API endpoints and calls the respective service layer functions.
- **Service Layer:** Handles the business logic implementation. Includes interfaces for the student service, and implementations extend these interfaces. Logs are created to denote successes and failures.
- **Repository Layer:** Handles all interaction with the database. Uses JPA for easy database operations.

3.2.4 Testing

Testing is performed in the following way using JUnit to automate the tests:

- Create unit test cases for each method in the repository layer. Verify that data access operations (CRUD) are performed correctly using an in-memory H2 database.
- Create unit test cases for each method in the service layer. Mock the repository layer using Mockito to isolate the business logic and test different scenarios.

- Create integration tests to test the API endpoints provided by the controller using mockMVC. Verify that the endpoints handle requests and responses correctly for different HTTP methods (GET, POST, PUT, DELETE).

3.3 Book Lending Application

3.3.1 Functionality

- There are predefined admins (with admin credentials) who can add and delete librarians (with librarian credentials).
- Librarians can then log in to the application and access two pages:
  - **Student Page:**
    - The user can select a student from a list of students, and also add, update, and delete students (APIs from the Student App are queried).
    - The user can then view the books borrowed by the student, issue a book to the student, and return a book borrowed by the student.
  - **Books Page:**
    - The user can select a book from a list of books, and also add, update, and delete books (APIs from the Books App are queried).
    - The user can then view the student who borrowed the book, return the book, and issue the book to a student.

3.3.2 Auth Endpoints

The following are the API endpoints for all user authentication and authorization (Admin and Librarian):

Method	**Endpoint**	**Description**
GET	/api/admin/getLibrarian	Returns a list of librarians, i.e., users with the role LIBRARIAN. Only a user with ADMIN role can access this route.
POST	/api/admin/addLibrarian	Adds a librarian, i.e., a user with the role LIBRARIAN. Only a user with ADMIN role can access this route.
DELETE	/api/admin/deleteLibrarian/`email`	Deletes the librarian with the given email id. Only a user with ADMIN role can access this route.
POST	/api/users/signup	Adds a user. Everyone has access to this route.

Method	Endpoint	Description
POST	/api/users/login	Verifies the credentials of a user and returns a JWT token, which can be used to access remaining endpoints. Everyone has access to this route.
GET	/api/users/logout	Logs out the user. Everyone has access to this route.

3.3.3 Auth Project Structure

To implement authentication, the project uses the Spring Security Maven dependency for authentication and authorization of users. When a request is received by the backend, before executing the service code of the route, the auth layer intercepts the request and, using the JWT token, retrieves the user credentials.

- If the credentials are valid, the role of the user is checked. This is implemented using separate filters, where the first filter checks for the validity of the user, and the second filter checks the role.
- The role is used to verify if the user has authorization to access a route.
- The implementation is found in the `config` directory, where the `SecurityConfig` file contains the necessary filters. The filters are implemented in the same directory.
- A `JWTService` class is used in the filters to handle the generation of JWT tokens and to retrieve user credentials.

3.3.4 Student Endpoints

The following are the API endpoints for performing CRUD operations on students in the application. These APIs query external APIs from the Student App:

Method	Endpoint	Description
GET	/api/booklending/students	Returns all students added to the database.
GET	/api/booklending/students/rollNo/`rollNo`	Returns the student corresponding to the roll number.
POST	/api/booklending/students	Takes a student object and adds it to the database.
PUT	/api/booklending/students	Updates the student in the database corresponding to the roll number in the request body.
DELETE	/api/booklending/students/rollNo/`rollNo`	Deletes the student corresponding to the roll number from the database.

3.3.5 Book Endpoints

The following are the API endpoints for performing CRUD operations on books in the application. These APIs query external APIs from the Books App:

Method	Endpoint	Description
GET	/api/booklending/books	Returns all books added to the database.
GET	/api/booklending/books/code/`code`	Returns the book corresponding to the code.
POST	/api/booklending/books	Takes a book object and adds it to the database.
PUT	/api/booklending/books	Updates the book in the database corresponding to the code in the request body.
DELETE	/api/booklending/books/code/`code`	Deletes the book corresponding to the code from the database.

3.3.6 Book Lending Endpoints

We have a `BookLendingEntity` class with the following fields:

- transactionId
- rollNo (of the student)
- bookCode (of the book)
- issued (boolean: true if the book is issued, false if the book has been returned)
- issueDate
- returnDate

The following are the API endpoints for lending and returning books to students, as well as viewing the books issued by a student and the student who borrowed a book:

Method	Endpoint	Description
POST	/api/booklending/lendBook	Issues a book to a student. Sets `issued` to true, `issueDate` to today, and `returnDate` to null. Saves the book lending entity. Returns an error if the student or book does not exist or if the book is already lent.

Method	Endpoint	Description
PUT	/api/booklending/returnBook/`transactionId`	Returns the book corresponding to the transaction id. Sets `issued` to false and `returnDate` to today.
GET	/api/booklending/getBook/`rollNo`	Returns all books borrowed by the student with the given roll number.
GET	/api/booklending/getStudent/`code`	Returns the student who borrowed the book corresponding to the code.

3.3.7 Frontend Integration

The frontend queries the APIs above and displays the information accordingly. It provides an interface for admins to manage librarians and for librarians to manage students and books.

3.4 Frontend

3.4.1 Frontend Flow

The following flow outlines the actions a user can perform within the Book Lending Application frontend:

1. When the application's base URL is opened, the login page is displayed.
2. A user can log in as either an admin or a librarian:
   (a) Admin: Can add, remove, and view librarians.
   (b) Librarian: Can navigate to the books or students page:
      i. **Books Page:** View, add, update, and delete books. The librarian can also view the student who has borrowed a specific book and allow them to return it.
      ii. **Students Page:** View, add, update, and delete students. The librarian can also view the books borrowed by a specific student and manage the return of borrowed books.

3.4.2 Pages Overview

The application contains several pages that render different functionalities for the users. Below is a description of the important pages:

Page	Path	Description
Login Page	/login	Renders the login form where users can input their credentials. Upon successful login, the user is redirected to the appropriate dashboard (Admin or Librarian).
Admin Dashboard	/adminlibrarian	Displays the admin interface, allowing admins to add, remove, and view librarians.
Librarian Dashboard	/dashboard	Allows the librarian to choose between the books page or students page.
Books Page	/books	Displays all books in the system. Allows the librarian to view, add, update, delete, and search for books.
Book Details Page	/bookdetails/:id	Displays details of a specific book based on its ID. Allows the librarian to view the student who has borrowed the book and allow them to return it.
Students Page	/students	Displays all students in the system. Allows the librarian to view, add, update, delete, and search for students.
Student Details Page	/studentdetails/:id	Displays details of a specific student based on their roll number. Allows the librarian to view the books borrowed by the student and return them if necessary.

4 Version Control

We used Git and GitHub to maintain version control. We also maintained several branches to implement different features.

5 CI and CD (Jenkins)

We had a Jenkinsfile that acheives the following functionality:

- **Environment Variables:**
  - `DOCKERHUB_CRED`: Credentials for DockerHub.
  - `GITHUB_REPO_URL`: URL of the GitHub repository.
  - `VAULT_CRED`: Credentials for Ansible Vault.
- **Agent:**
  - Uses any available agent to run the pipeline.
- **Stages:**

- **Stage 1: Git Clone**
  - Clones the specified branch (`kalyankube`) from the GitHub repository.
- **Stage 2: Docker Compose Build**
  - Builds Docker containers using `docker-compose`.
- **Stage 3: Push to DockerHub**
  - Logs into DockerHub using credentials.
  - Pushes the built Docker images to DockerHub.
- **Stage 4: Clean Unwanted Docker Images**
  - Cleans up unwanted Docker images, volumes, and orphan containers using `docker-compose down`.
- **Stage 5: Prepare Docker Compose File**:
  - Archives the `infra` directory and `my_vault.yml` file.
  - Prints the workspace location and lists its contents.
- **Stage 6: Ansible Deployment**
  - Disables Ansible host key checking.
  - Creates a temporary file for the Vault password.
  - Runs the Ansible playbook `Deploy-ubuntu.yml` with the workspace path and Vault password.
  - Removes the temporary Vault password file.

- **Post Actions:**
  - **Success:**
    - Prints a success message.
  - **Failure:**
    - Prints a failure message.
    - Cleans up Docker resources using `docker-compose down`, ignoring errors.

The project was configured with Git SCM polling and ngrok.

6 Continuous Delivery (Ansible)

This project uses ansible to deploy the application on the host machine. Here are the steps that are involved :

1. **Roles** : For this project we have made use of ansible roles to modularize the various components of the deployment. This includes the following :

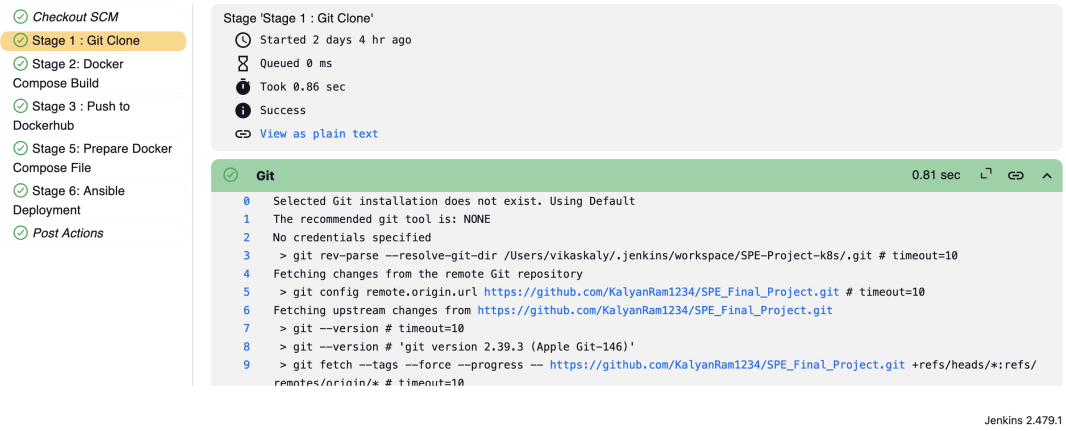

Figure 1: Jenkins Build

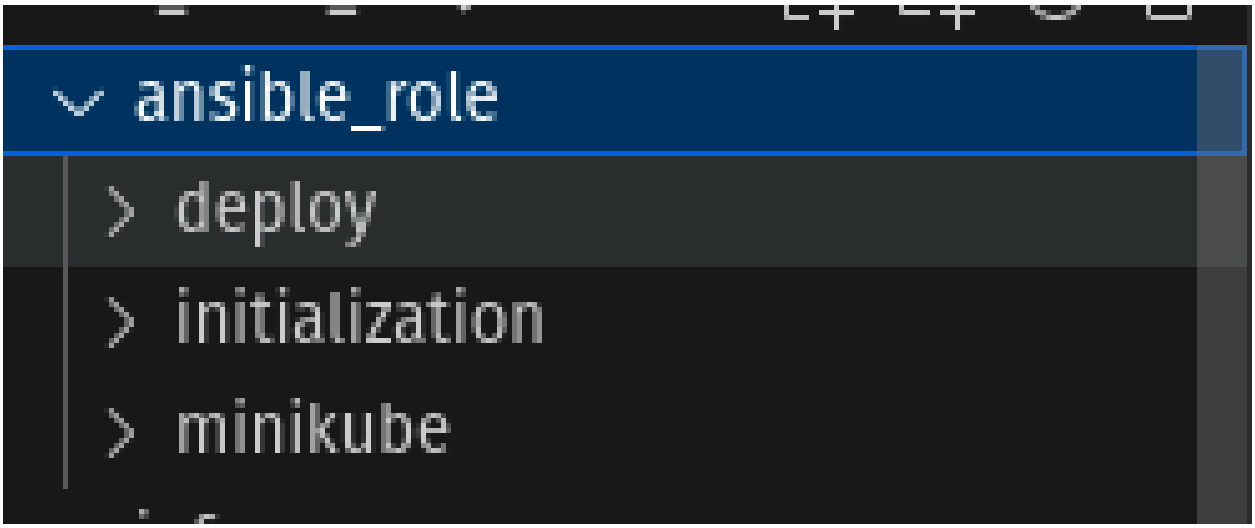

Figure 2: Ansible Roles

(a) **Initialization** : This involves copying all the kubernetes files to ensure that the host has all the files required to run the application. This is passed by using the jenkins artifacts feature where the files are added to the artifacts which can then be accessed by the ansible environment.

(b) **Minikube Setup** : Here we start minikube by using the command **minikube start –driver=docker**. We also check if minikube is already up and running, if so we skip the minikube start step.

(c) **Deployment**: Here we run the cleanup.sh script to remove existing deployments of the application. Then we run the deploy.sh script to run all the services and deployments.

2. **Vaults** : We have used the ansible vaults to pass the ssh password in a encrypted manner. The vault password is stored in the Jenkins credentials manager and is used

```yaml
- name: Check Minikube Status
  shell: minikube status | grep -i "running"
  register: minikube_status
  ignore_errors: true  # Ignore errors if Minikube isn't running

- name: Start minikube
  shell: minikube start --driver=docker
  when: minikube_status.rc != 0
```

Figure 3: Use of roles

```
$ANSIBLE_VAULT;1.1;AES256
35646332316331343633363734363465313232393463646136643838316630353364613339623336
62343061623433626338383666633535373933306431323 60a6533643137626331353038666661 64
64326461353266663332323431366363643835633738613839373763336235633062386532656362
38616466343430343 10a37346330343933386163639316364626330326337393866366136663266
31666166323533363138663930383135623961623261346439626636633430653336
```

Figure 4: Vault File Encryption

while invoking the ansible playbook command to ssh into the host environment.

7 Containerization (Docker)

The library management project was containerized using Docker to ensure seamless deployment and portability of the application across different environments. The project involved multiple services, each containerized individually with Dockerfiles for the backend and frontend. Additionally, a Docker Compose configuration was used to **build** all the services collectively.

7.1 Backend Containerization

For each backend service (*Book Application*, *Student Application*, and *Book Lending Application*), the following Dockerfile configuration was used:

- Base image: `maven:3.8.4-openjdk-17-slim` for optimal performance and compatibility with Java-based Spring Boot applications.
- Working directory: `/app` to house the application files.
- Dependencies: `pom.xml` file was copied first, and Maven's `go-offline` command was executed to pre-fetch all dependencies, improving build efficiency.

- Application code: The `src` folder was copied to the container to include the source code.
- Startup command: `mvn spring-boot:run` to test and launch the Spring Boot application directly.

7.2 Frontend Containerization

The frontend service was containerized with the following steps:

- Base image: `node:20-alpine` to leverage a lightweight Node.js environment.
- Working directory: `/app` to store the application files.
- Dependencies: `package.json` file was copied, and `npm install` was run to install all required dependencies.
- Application code: The entire frontend source was copied to the container.
- Startup command: `npm run dev` to start the Vite development server.

8 Orchestration and Scaling (K8s)

This project includes various Kubernetes configurations for deploying and managing multiple applications and services. Here is an overview of the different configurations:

1. **Deployments**:
   - `bookapp-infra/bookapp-depl.yaml`: Defines a deployment for the `bookapp` application.
   - `studentapp-infra/studentapp-depl.yaml`: Defines a deployment for the `studentapp` application.
   - `booklendingapp-infra/booklendingapp-depl.yaml`: Defines a deployment for the `booklendingapp` application.
   - `frontend-infra/libraryFrontend-depl.yaml`: Defines a deployment for the `library-frontend` application.
   - `prometheus/prometheus-deployment.yaml`: Defines a deployment for the Prometheus monitoring server.

   The deplyoments for the *bookapp, studentapp, booklendingapp and frontend* include memory configurations as well to ensure that the pod does not go out of memory. Every deplyoment only has 1 pod due to our machine's memory constraints.

2. **Services**:
   - `bookapp-infra/bookapp-svc.yaml`: Defines a service for the `bookapp` application.

- `booklendingapp-infra/booklendingapp-svc.yaml`: Defines a service for the `booklendingapp` application.
- `frontend-infra/libraryFrontend-svc.yaml`: Defines a service for the `library-frontend` application.
- `db-infra/mysql-service.yaml`: Defines a service for the MySQL database.

3. **Ingress**:
   - `ingress-infra/bookapp-ingress.yaml`: Defines an ingress for the `bookapp` application.
   - `ingress-infra/booklendingapp-ingress.yaml`: Defines an ingress for the `booklendingapp` application.
   - `ingress-infra/frontend-ingress.yaml`: Defines an ingress for the `library-frontend` application.
   - `ingress-infra/studentapp-ingress.yaml`: Defines an ingress for the `studentapp` application.
4. **StatefulSets**:
   - `db-infra/mysql-statefulset.yaml`: Defines a StatefulSet for the MySQL database.
5. **Horizontal Pod Autoscalers (HPA)**:
   - `hpa.yaml`: Defines HPAs for `bookapp`, `studentapp`, `booklendingapp`, and `library-frontend` applications to automatically scale the number of pods based on CPU utilization.
6. **Configuration for Promtail and Loki**:
   - `promtail-overrides.yaml`: Contains configuration overrides for Promtail, including server settings, client configurations, and scrape configurations.
   - `loki-distributed-overrides.yaml`: Contains configuration overrides for Loki, including server settings, storage configurations, and ingress settings.
7. **Kubernetes Secrets**

   The `app-secrets.yaml` file contains Kubernetes secrets for securely storing sensitive information related to the application and MySQL database. It includes Base64 encoded values for the application admin email, password, and username, as well as the MySQL database name, root password, and root username. These secrets are used to manage access and credentials securely within the Kubernetes cluster.
8. **Scripts**:
   - `deploy.sh`: A script to deploy all the configurations to the Kubernetes cluster.
   - `cleanup.sh`: A script to clean up the deployments and services from the Kubernetes cluster.

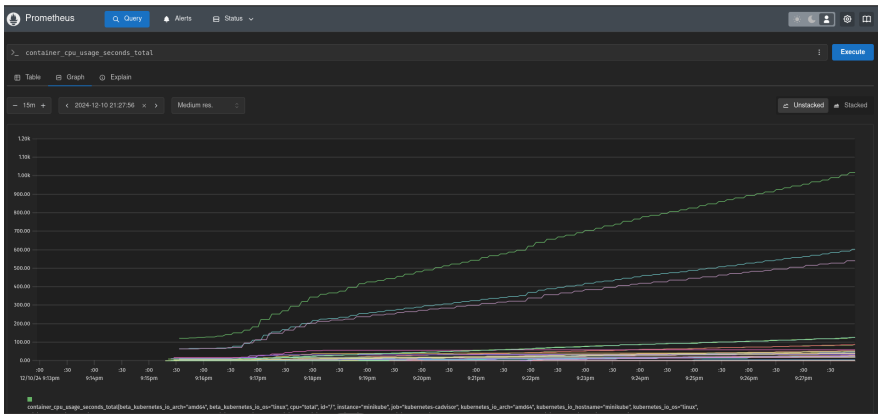

Figure 5: Container cpu utilization

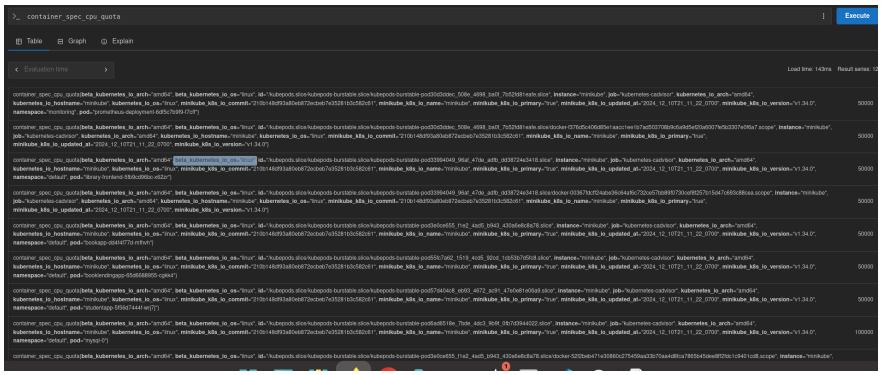

Figure 6: Container Memory Usage

- `loki.sh`: A script to install and configure Loki and Promtail using Helm.

These configurations collectively manage the deployment, scaling, and monitoring of the applications and services in the Kubernetes cluster.

9 Monitoring and Logging

For monitoring and logging we have use Grafana as the common dashboard tool and use Prometheus for Monitoring and Metrics, and used Loki for Logging. Loki will extract all the logs from all the namespaces and containers. Prometheus provides metrics on all pods, services, deployments and provides visualizations to monitor the performance of the application.

Due to system limitations we can only run one of the services at a time. But with enough resources it is very easy to create a common dashboard where we just have to specify the server urls as datasources which we have.

Here are some visualizations and logs that are generated by prometheus and loki :

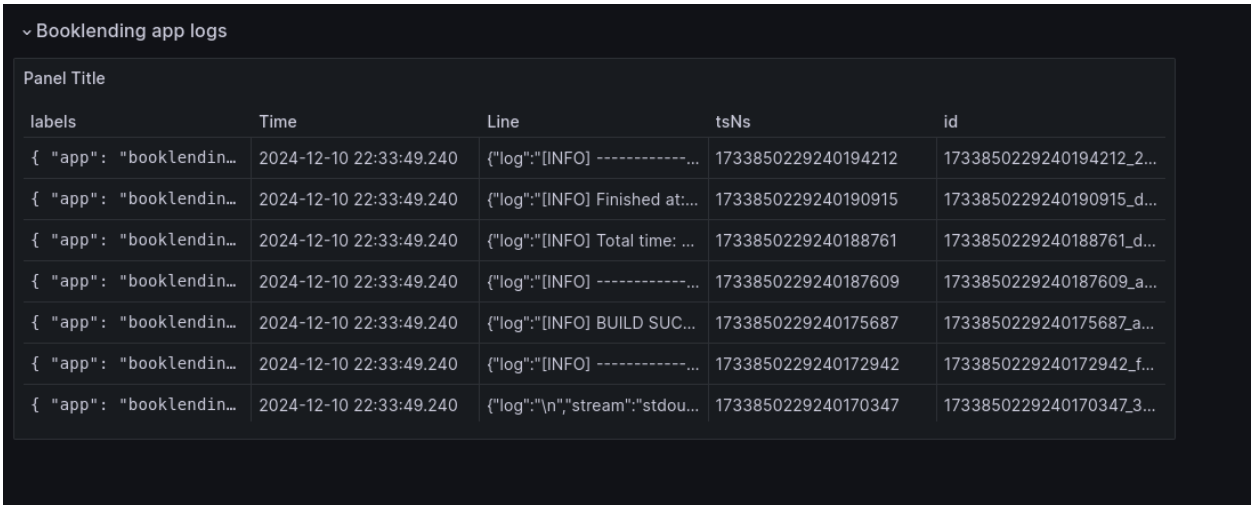

Figure 7: BookLending App Logs

10 Innovative Solution

As a part of the project implementation, we realized that starting minikube was a very timing taking and cpu intensive process. Additionally it is not required to start minikube everytime an update is made to the code. Hence we are checking the minikube status on the host and only run the command if there is no instance of minikube running on the host.

This also provides the additional benefit of not having to restart independent services like logging and monitoring that can run on their own as long as the datasource remains the same.