Library Management System

Department of Computer Science and Engineering
Indian Institute of Information Technology Sonepat

Submitted By
Siddharth Singh
12211026
&
Sankhadeep Roy Chowdhury
12211059

Submitted To
Dr. Vinay Pathak
(Assistant Professor)

INDEX

Report

Project Objectives – Library Management System for IIIT Sonepat

- Streamline Library Operations: Develop a lightweight and efficient system to manage book inventory, student records, and borrowing activities, reducing manual effort and errors.
- Enhance User Accessibility: Provide an intuitive and responsive frontend using JavaScript to ensure seamless interaction for students and staff across devices.
- Leverage FastAPI for Performance: Utilize FastAPI for building high-performance backend APIs, enabling rapid data processing and real-time updates.
- Secure and Scalable Architecture: Implement secure endpoints and structured integration to support future scalability and data integrity.
- Institution-Specific Customization: Tailor the system functionalities to meet the academic and administrative requirements of IIIT Sonepat.

Reason for Title Selection

The title *"Library Management System for IIIT Sonepat"* was chosen to clearly reflect the project's purpose and scope. It signifies that the system is not a generic solution but a customized platform tailored to the operational workflows and institutional context of IIIT Sonepat. The title emphasizes both the domain (library management) and the target environment (IIIT Sonepat), making it precise and relevant.

Comparative Analysis of Library Management Systems

Feature / System	LMS for IIIT Sonepat	Koha	Evergreen	SLiMS (Senayan Library Management System)
Technology Stack	FastAPI (Python), JavaScript	Perl, MySQL, Apache	Perl, PostgreSQL	PHP, MySQL
Performance	Lightweight, fast, low latency	Reliable, moderate performance	Scalable, optimized for consortia	Decent performance for small to mid-size use
User Interface	Custom-built, modern, responsive	Outdated but functional	Functional, less intuitive	User-friendly with simple UI
Customization	Highly customizable for IIIT Sonepat	High (requires Perl expertise)	Moderate (complex to configure)	High (PHP-based, easy to modify)
Deployment Complexity	Simple and modular	Moderate to high	High (requires advanced setup)	Low
Target Audience	Academic institutions like IIIT Sonepat	Public, academic, and special libraries	Large-scale library networks	Small to mid-sized libraries
Scalability	Scalable with FastAPI & async support	Scalable for large databases	Very scalable, multi-branch support	Limited scalability
API Support	Full RESTful API (FastAPI native)	Yes (REST API available)	Yes (limited REST endpoints)	Limited API support
Documentation	Project-specific, lightweight docs	Extensive community documentation	Detailed official and community docs	Moderate documentation

Team Contribution

The development of the *Library Management System for IIIT Sonepat* was the result of dedicated collaboration between two contributors: **Sankhadeep Roy Chowdhury (12211059)** and **Siddharth Singh (12211026)**. Their focused efforts and technical proficiency were instrumental in bringing this project from concept to completion, overcoming significant challenges along the way.

- **Sankhadeep Roy Chowdhury (12211059)**
  Sankhadeep took full responsibility for designing and implementing the **backend architecture** using **FastAPI**. He meticulously developed and optimized a robust set of RESTful API endpoints to manage books, student data, authentication, and library transactions. His work ensured a secure, high-performance foundation for the system. Navigating the complexities of asynchronous data handling and security protocols, Sankhadeep delivered a scalable and reliable backend that serves as the backbone of the application.

- **Siddharth Singh (12211026)**
  Siddharth led the **frontend development and backend integration**, ensuring seamless interaction between the user interface and server-side components. His contributions enabled a smooth, responsive, and intuitive user experience across platforms. Additionally, Siddharth was responsible for writing comprehensive **technical documentation**, which reflects his clarity of thought and attention to detail. His ability to bridge design and functionality played a pivotal role in the system's success, especially when integrating asynchronous endpoints with real-time front-end feedback—a notably complex task.

Both contributors have demonstrated exceptional problem-solving skills, perseverance, and a deep understanding of full-stack development. Their coordinated efforts resulted in a lightweight yet powerful solution, tailored specifically to the needs of IIIT Sonepat. The project stands as a testament to their technical excellence and commitment to quality.

Approach and Techniques Used-

Aspect	How It Was Implemented
Backend Development	The backend was developed using **FastAPI**, chosen for its high performance and support for asynchronous programming. Each endpoint was modularized based on functionality (e.g., book management, student records, authentication), ensuring maintainability and scalability. FastAPI's decorator-based routing made it easier to map HTTP methods to functions cleanly.
Frontend Integration	A custom frontend was built using **JavaScript**, integrated with the backend through fetch() or axios calls. Event-driven functions allowed dynamic rendering of data (e.g., student list, book availability). The integration was tightly coupled with backend responses, ensuring real-time updates and a smooth user experience.
Database Interaction	Data storage and retrieval were handled using **SQLAlchemy ORM**, which abstracted raw SQL queries and allowed object-oriented interaction with the database. Tables were created for books, students, and transactions, with relationships and constraints enforced to maintain data integrity.
Authentication & Security	Secure user authentication was implemented using **JWT (JSON Web Tokens)**. When users log in, a token is issued and required for all further requests to protected endpoints. Middleware was added to validate tokens, and proper headers were configured to ensure Cross-Origin Resource Sharing (CORS) security during frontend-backend communication.
Error Handling & Validation	Input data from users was validated using **Pydantic models**, which ensured correct data types and required fields. Structured try-except blocks were used for handling server errors, and descriptive JSON responses were returned to the frontend for user-friendly feedback.
API Documentation	**FastAPI's built-in Swagger UI** automatically generated interactive API documentation. This allowed developers to test endpoints in real time, view expected input/output formats, and reduce onboarding time for future contributors.

Aspect	How It Was Implemented
Deployment Strategy	The project was designed to be **lightweight and environment-independent**, making it easy to deploy on local servers, college intranet, or cloud platforms. Environment variables were used for sensitive configurations, ensuring secure and portable deployments.
Version Control & Collaboration	The team used **Git and GitHub** to manage code collaboratively. Branches were created for individual features, with pull requests and code reviews ensuring quality control. This also enabled version history, backups, and rollback when necessary.
Documentation	Detailed **technical documentation** was written covering system architecture, API references, installation steps, and usage guides. Markdown files and in-code docstrings were used to ensure that both users and future developers could understand and maintain the project effectively.

Pseudo Code:-

1.User Authentication

```python
from fastapi import APIRouter, HTTPException, Depends, Response, status
from fastapi.security.oauth2 import OAuth2PasswordRequestForm
from sqlalchemy.ext.asyncio import AsyncSession
from config.database import get_db
from sqlalchemy.future import select
from schemas import auth, token
from utils import verify, create_access_token, send_otp, get_current_user
import model
 Ctrl+L to chat, Ctrl+K to generate
router = APIRouter(
    tags=["Authentication"]
)

@router.post('/login', response_model=token.Token)
async def login(userCredentials : OAuth2PasswordRequestForm = Depends(), db: AsyncSession = Depends(get_db)):
    result = await db.execute(select(model.User).where(model.User.username == userCredentials.username))
    user = result.scalar_one_or_none()
    if not user or not verify(userCredentials.password, user.password):
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Invalid Credentials")

    token = create_access_token(data={"username" : user.username, "role" : user.role, "jobId" : user.jobId})

    return {"token_type" : "bearer", "token" : token}


@router.post('/otp')
async def get_otp(auth: auth.userOTP, client: int = Depends(get_current_user)):
    otp = await send_otp(auth.email, auth.name)
    if otp is None:
        raise HTTPException(status_code=500, detail="OTP not sent")
    return {"OTP" : otp}
```

2.User Module

```python
router = APIRouter(
    prefix= "/users",
    tags=['Users']
)


@router.get("/", response_model= List[user.baseUser])
async def get_users(db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    # result = await db.execute(text('SELECT * FROM user'))
    # users = result.mappings().all()
    result = await db.execute(select(model.User))
    users = result.scalars().all()
    return users



@router.get("/{name}", response_model=user.baseUser)
async def get_user(name : str, db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    # result = await db.execute(text("SELECT * FROM user WHERE id = :id"), {"id": id})
    # user = result.mappings().first()
    result = await db.execute(select(model.User).where(model.User.username == name))
    user = result.scalar_one_or_none()
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    return user



@router.post("/", status_code=status.HTTP_201_CREATED, response_model=user.baseUser)
async def create_user(user : user.createUser, db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    if(client.role == False):
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Unauthorized access")
```

3.Student Module

```python
router = APIRouter(
    prefix= "/students",
    tags=['Students']
)


@router.get("/", response_model= List[student.baseStudent])
async def get_students(db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    result = await db.execute(select(model.Student))
    students = result.scalars().all()
    return students

@router.get("/{rollNo}", response_model=student.baseStudent)
async def get_student(rollNo : str, db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    result = await db.execute(select(model.Student).where(model.Student.studentId == rollNo))
    student = result.scalar_one_or_none()
    if not student:
        raise HTTPException(status_code=404, detail="Student not found")
    return student
Ctrl+L to chat, Ctrl+K to generate
@router.post("/", status_code=status.HTTP_201_CREATED, response_model=student.baseStudent)
async def create_student(student : student.createStudent, db: AsyncSession = Depends(get_db), client : int = Depends(get_current_use
    new_student = model.Student(
        firstName=student.firstName,
        lastName=student.lastName,
        email=student.email,
        phone=student.phone,
        department=student.department,
        batch=student.batch,
        studentId=student.studentId
    )

    db.add(new_student)
    await db.commit()
```

4.Book Module

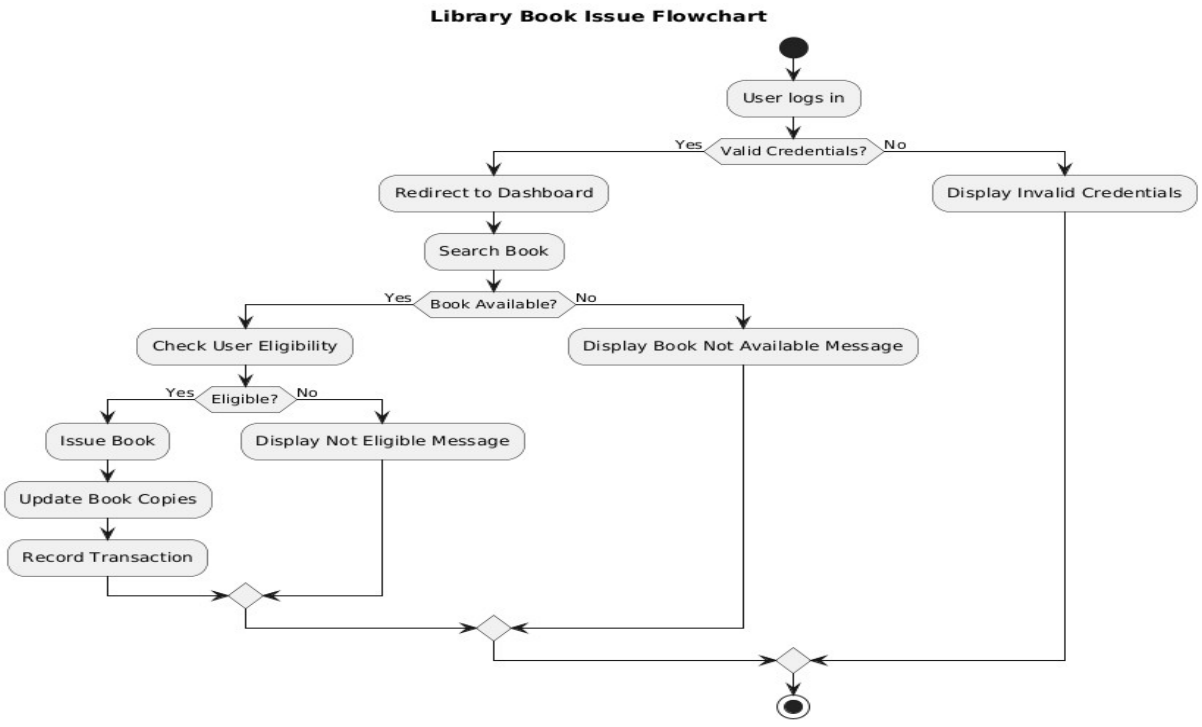
```python
router = APIRouter(
    prefix= "/books",
    tags=['Books']
)


@router.get("/", response_model= List[book.baseBook])
async def get_books(db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    result = await db.execute(select(model.Book))
    books = result.scalars().all()
    return books

@router.get("/available", response_model= List[book.baseBook])
async def get_books(db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    result = await db.execute(select(model.Book).where(model.Book.available > 0))
    books = result.scalars().all()
    return books


@router.get("/byName/{title}", response_model=List[book.baseBook])
async def get_book_byName(title : str, db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    result = await db.execute(select(model.Book).where(model.Book.title == title))
    book = result.scalars().all()
    if not book:
        raise HTTPException(status_code=404, detail="Book not found")
    return book


@router.get("/byISBN/{isbn}", response_model=book.baseBook)
async def get_book_byISBN(isbn : str, db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    result = await db.execute(select(model.Book).where(model.Book.isbn == str(isbn)))
    book = result.scalar_one_or_none()
    if not book:
```

5.BookIssue Module

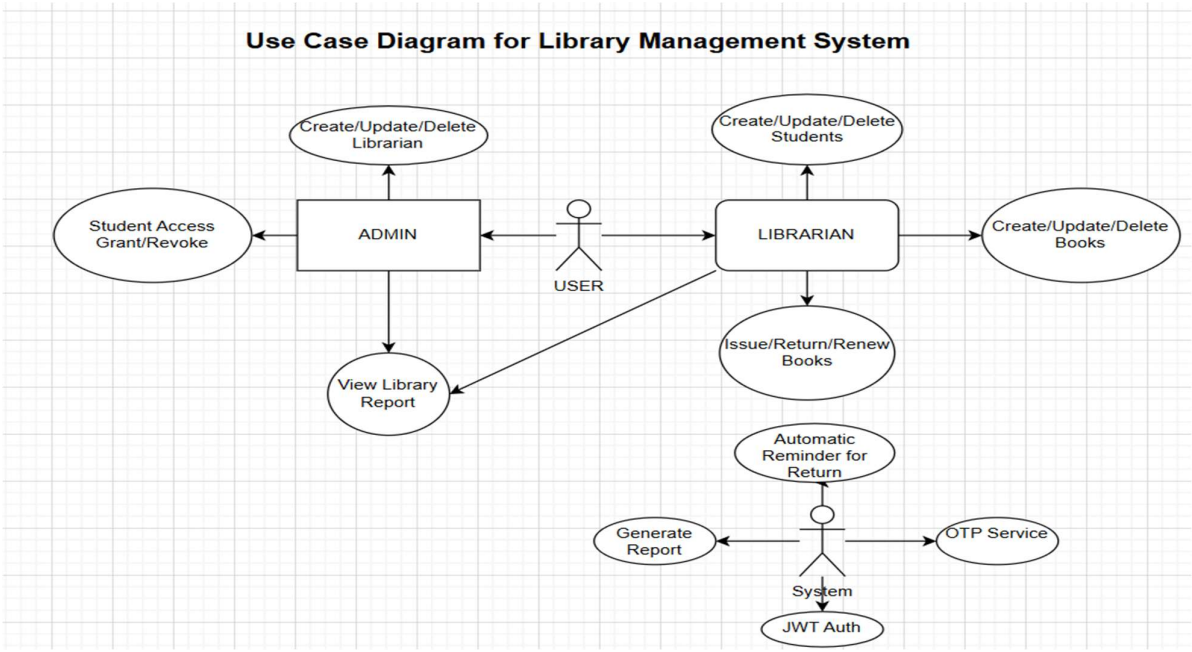
```python
router = APIRouter(
    prefix= "/issues",
    tags=['IssueBooks']
)

@router.get("/", response_model= List[bookIssue.baseBookIssue])
async def get_issues(db: AsyncSession = Depends(get_db), client : int = Depends(get_current_user)):
    result = await db.execute(select(model.Issue))
    issues = result.scalars().all()
    return issues

@router.get("/search")
async def get_issues(student_id: str, book_id: str, db: AsyncSession = Depends(get_db), client: int = Depends(get_current_user)):
    query = select(model.Issue).where(
        model.Issue.student_id == student_id,
        model.Issue.book_id == book_id
    )
    result = await db.execute(query)
    return result.scalars().all()
Ctrl+L to chat, Ctrl+K to generate
@router.post("/", status_code=status.HTTP_201_CREATED)
async def create_issue(issue: bookIssue.createBookIssue, db: AsyncSession = Depends(get_db), client: dict = Depends(get_current_user
    query = select(model.Issue).where(
        model.Issue.book_id == issue.book_id,
        model.Issue.student_id == issue.student_id,
        or_(model.Issue.status == "Issued", model.Issue.status == "Renewed")
    )
    result = await db.execute(query)
    issue_exists = result.scalars().first()
    if issue_exists:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Book already issued to the student")

    count_query = select(func.count()).where(
```

Schematics:

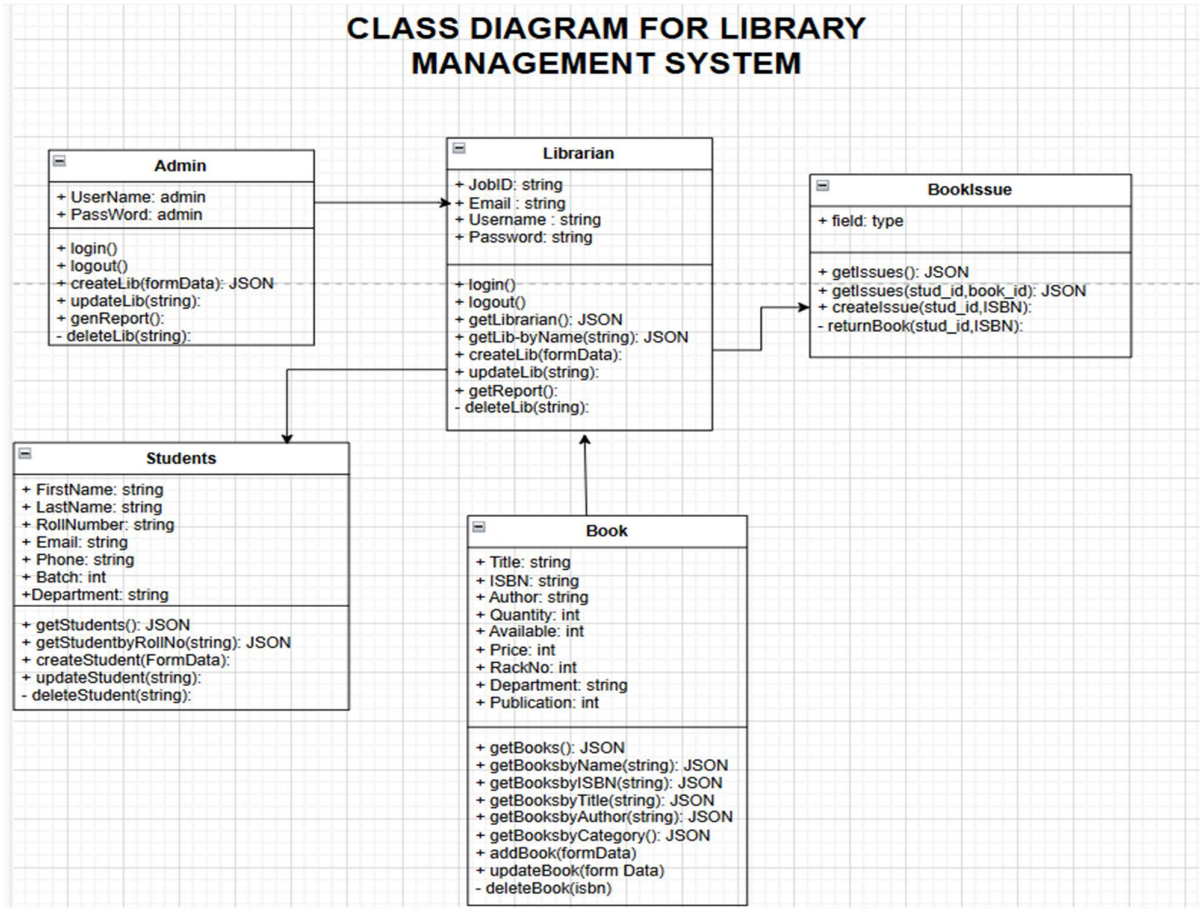
- **FlowChart:**



- **Use-Case Diagram:**



- **Class-Diagram:**

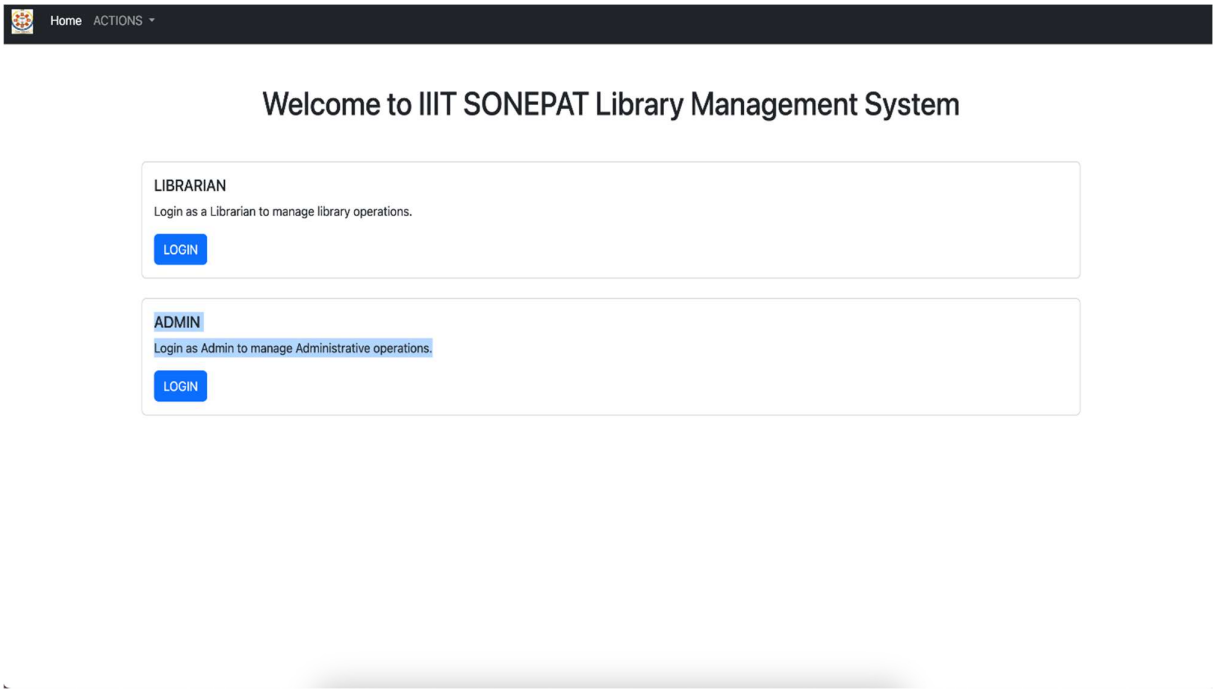


API Endpoints Summary

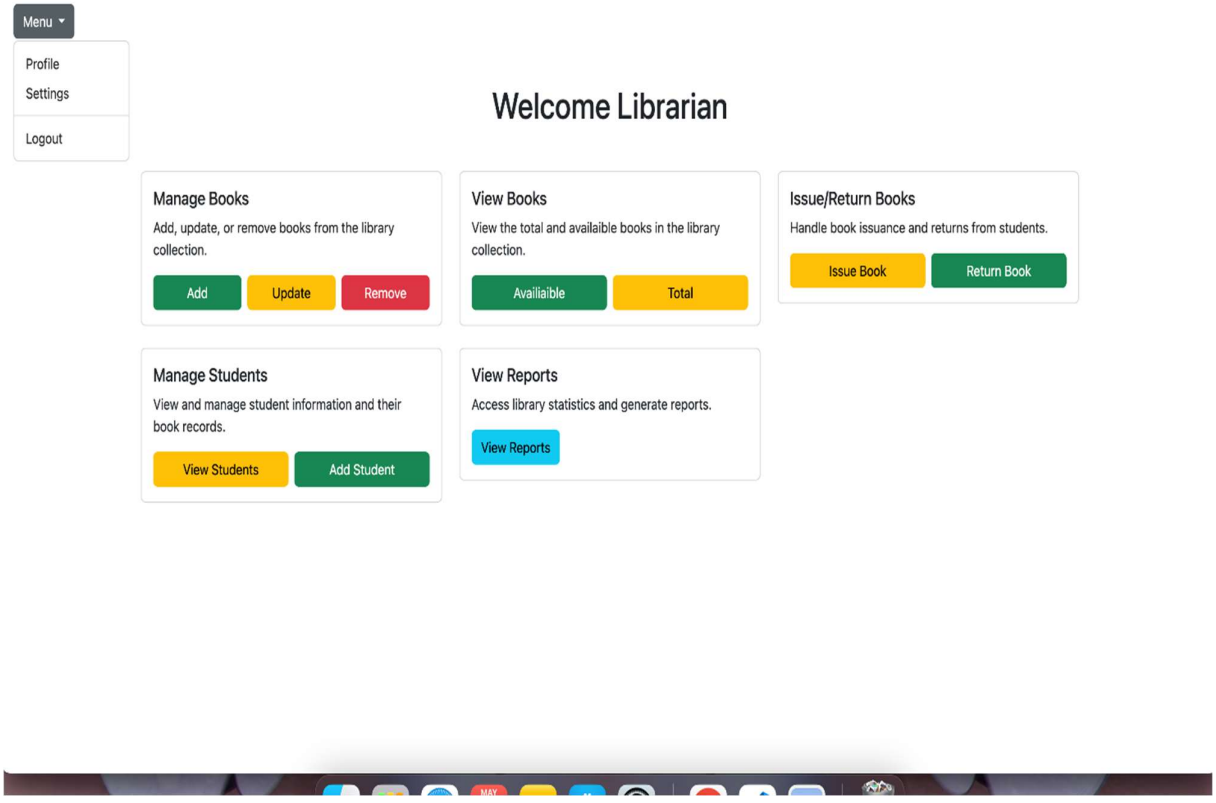
Controller	Endpoint	Method	Description	Access	Response
StudentController	/students/	GET	Fetch all students	Authenticated users	List of student records
	/students/{rollNo}	GET	Retrieve student details by roll number	Authenticated users	Single student object
	/students/	POST	Create a new student	Authenticated users	Newly created student record
	/students/{rollNo}	PUT	Update student details by roll number	Authenticated users	Updated student object
	/students/{rollNo}	DELETE	Delete a student record	Authenticated users	No Content (204)
UserController	/users/	GET	Retrieve all user records	Admin only	List of users
	/users/{username}	GET	Get details of a specific user by username	Admin only	Single user object
	/users/	POST	Create a new user	Admin only	New user object
	/users/	PUT	Update existing user	Admin only	Updated user object
	/users/{username}	DELETE	Delete user by username	Admin only	No Content (204)
BookController	/books/	GET	List all books	Authenticated users	List of book records
	/books/{isbn}	GET	Get book details by ISBN	Authenticated users	Single book object
	/books/	POST	Add a new book	Admin only	New book record
	/books/{isbn}	PUT	Update a book by ISBN	Admin only	Updated book object
	/books/{isbn}	DELETE	Delete a book by ISBN	Admin only	No Content (204)
BookIssueController	/issuedBooks/departmentWise	GET	Count of issued books, grouped by department	Authenticated users	List with department & count
	/dailyBookIssues	GET	Daily book issue count with corresponding date	Admin only	Date-wise issue list
AuthController	/login	POST	Authenticate user and generate JWT token	Public	JWT access token
	/	GET	Get current authenticated user’s details	Authenticated users	Current user object

Results

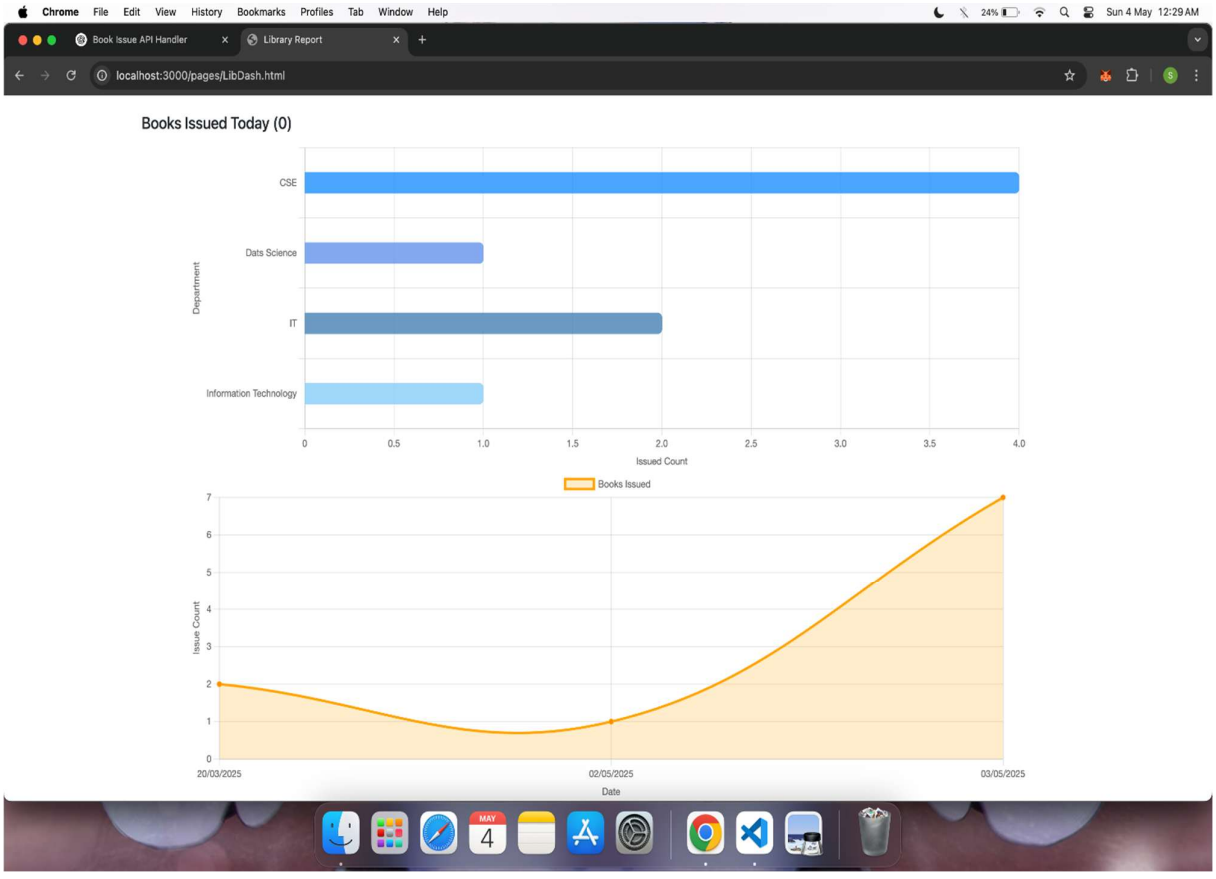
1. Landing-Page



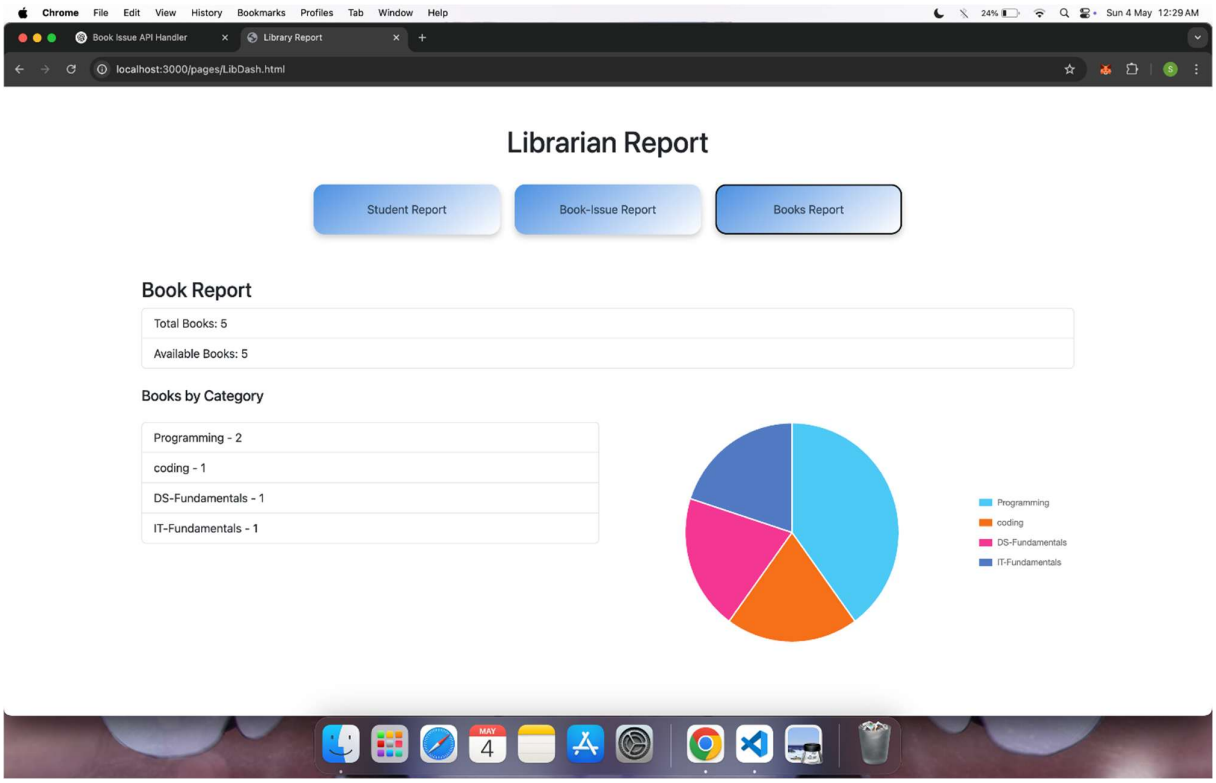
2. Librarian Dashboard



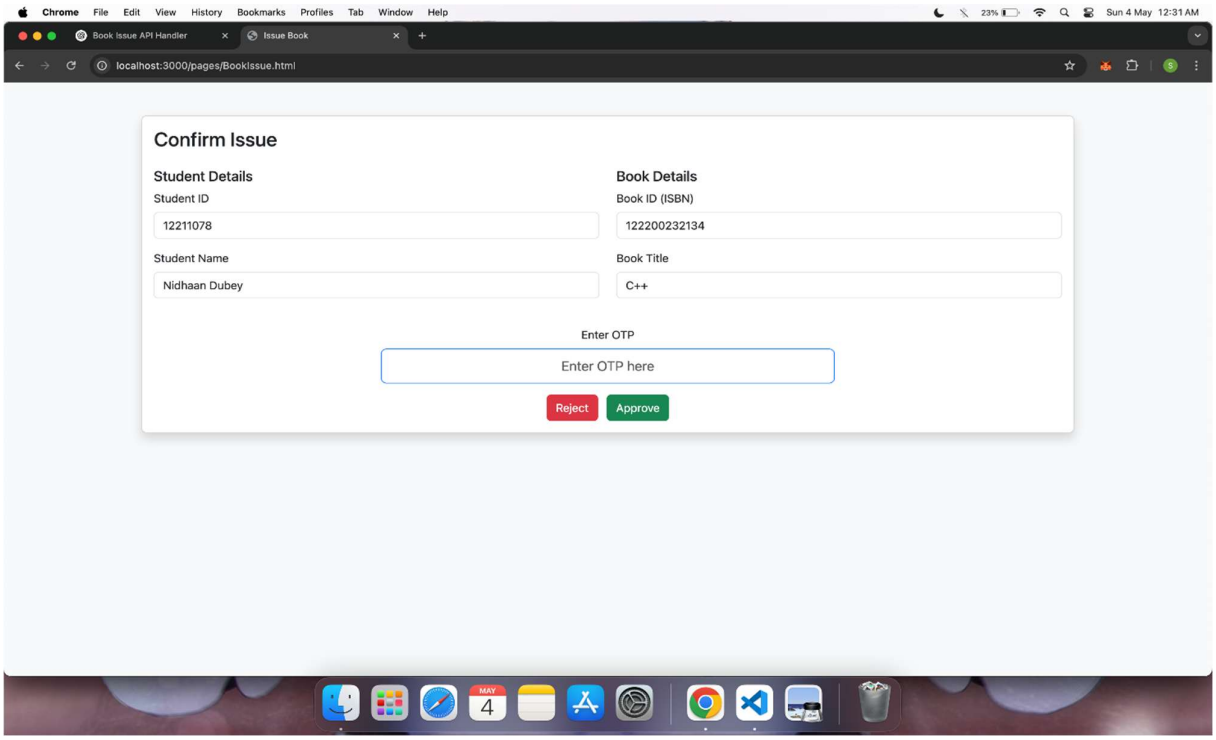
3. Book-Issue Report



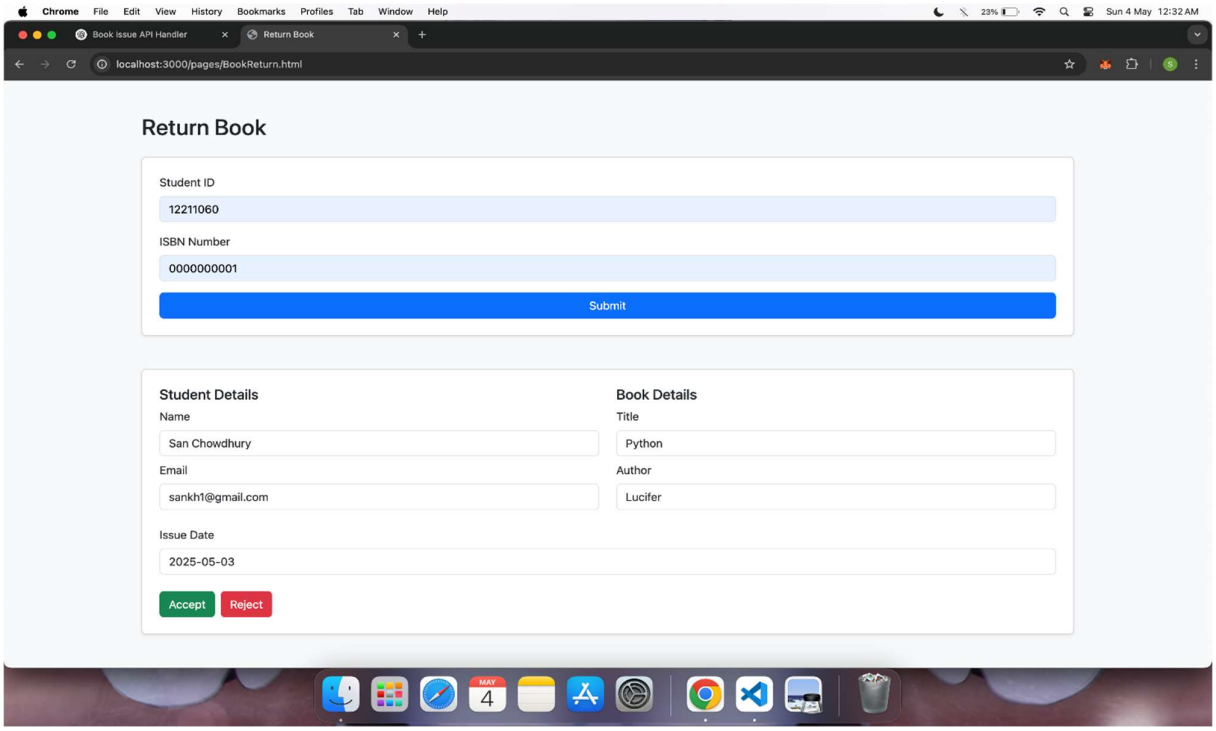
4. Book Report



5. Issuing Book



6. Return Book



Limitations:

- Limited Access Control:
The current system lacks granular access control features (e.g., roles such as librarian, admin, and student), which could be useful for further customization in the future. All users currently share similar privileges.

- Offline Functionality:
The system is designed to operate in an online environment, and does not offer offline access. If network connectivity is lost, users cannot access the system or make transactions.

- Scalability Concerns:
While the system is scalable to a reasonable extent, it is primarily designed for a medium-sized institution like IIIT Sonepat. Larger universities with higher traffic may need additional performance tuning and database optimizations.

- Complexity in Integration with External Systems:
The system does not support easy integration with third-party systems (e.g., external catalog databases, RFID systems, or university-wide authentication platforms) without substantial modifications.

-Limited Mobile Support:
While the frontend is responsive, it may not provide the best user experience on mobile devices, and additional optimization could be required for mobile-first use cases.

Future Work

- **Role-Based Access Control (RBAC):**
  Implement role-based access control to differentiate privileges between students, librarians, and administrators. This will provide more flexibility and security in user management.

- **Offline Mode:**
  Develop an offline mode for the system, allowing users to interact with the library management system even without an internet connection. Local data synchronization could be implemented when connectivity is restored.

- **Mobile Application Development:**
  Create a dedicated mobile application for both Android and iOS platforms, enhancing the user experience with native features such as push notifications and improved responsiveness for mobile devices.

- **AI-Driven Recommendations:**
  Integrate machine learning algorithms to provide book recommendations based on user activity and preferences. This would enhance user engagement and promote the discovery of books.

- **Enhanced Reporting and Analytics:**
  Develop more advanced reporting features to help the library administration track usage patterns, popular books, and resource allocation for more informed decision-making.

- **Integration with RFID and Barcode Systems:**
  Introduce RFID technology for automated book check-ins/outs and integration with barcode scanners to further streamline library operations.

References

1. **FastAPI Documentation**
   - "FastAPI: The modern, fast (high-performance) web framework for building APIs with Python 3.6+." FastAPI, https://fastapi.tiangolo.com/.
2. **SQLAlchemy Documentation**
   - "SQLAlchemy Documentation," SQLAlchemy, https://www.sqlalchemy.org/.
3. **JWT Authentication**
   - "JSON Web Tokens (JWT) Introduction," Auth0, https://auth0.com/docs/secure/tokens/json-web-tokens.
4. **Pydantic Documentation**
   - "Pydantic: Data validation and settings management using Python type annotations," https://pydantic-docs.helpmanual.io/.
5. **JavaScript Fetch API**
   - "Fetch API," Mozilla Developer Network (MDN), https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
6. **Koha Library Management System**
   - "Koha: Open-source Integrated Library System," Koha Community, https://koha-community.org/.
7. **Evergreen Library System**
   - "Evergreen: Open-source Library Software," Evergreen Project, https://evergreen-ils.org/.