Distributed Systems Project

Task 1: Design and Architecture
Library management system

Functional requirements of the system:

1. User authentication and authorization:
    - Register new users.
    - Authenticate users during login.
    - Manage user roles and permissions.
2. Book search:
    - Search for books by title, author, or genre.
    - Filter and sort search results.
    - Retrieve detailed information about a specific book.
3. Book availability and reservation:
    - Check the availability of a book.
    - Reserve a book for a user.
    - Cancel a reservation.
4. Book borrowing and returning:
    - Check out a book for a user.
    - Return a borrowed book.
    - Calculate and apply late fees for overdue books.
5. User profile management:
    - View and update user profile information.
    - View borrowing history.
    - Manage user preferences and settings.
6. Notifications:
    - Send notifications for overdue books.
    - Notify users when a reserved book becomes available.
    - Manage user notification preferences.
7. Inventory management:
    - Add new books to the library's inventory.
    - Update book information.
    - Remove books from the inventory.

Key services as stand-alone autonomous business units:

1. Authentication Service:
    - Scope: Handles user registration, authentication, and authorization.
    - Communication Pattern: RESTful API for communication with other services to verify user credentials and permissions.

2. Book Search Service:
    - Scope: Provides search functionality for books, including filtering and sorting results.

- Communication Pattern: RESTful API for communication with other services to retrieve book details and availability.

3. Book Availability and Reservation Service:
   - Scope: Manages book availability, reservations, and cancellations.
   - Communication Pattern: RESTful API for communication with other services to update book availability and notify users of reservation updates.

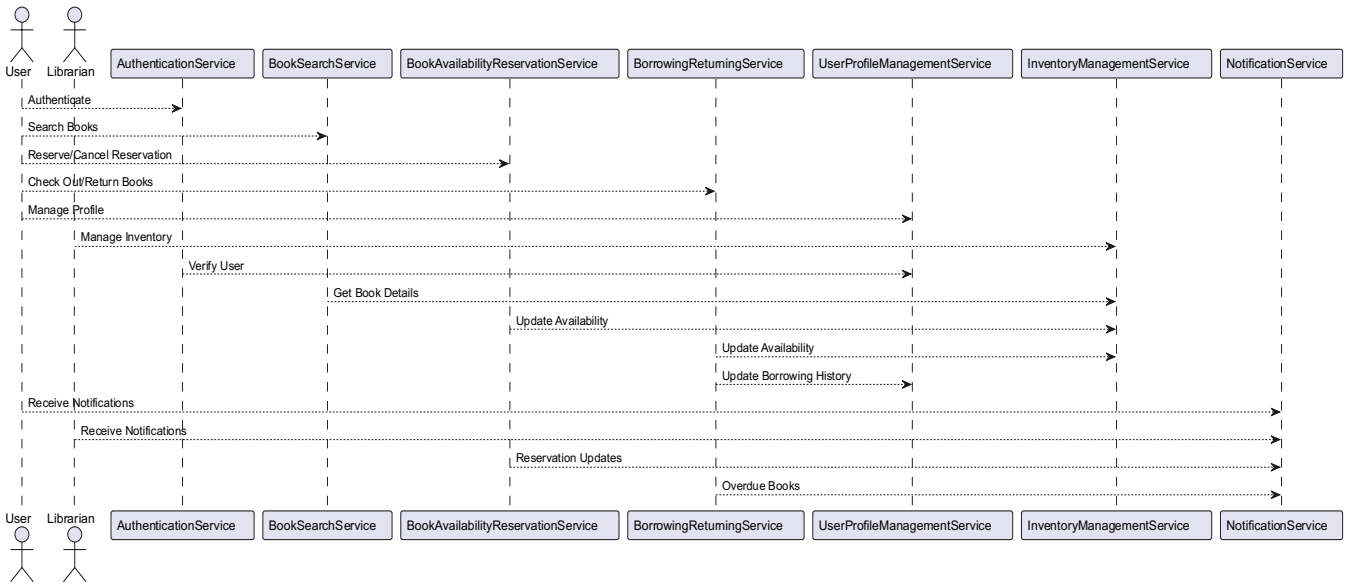
4. Borrowing and Returning Service:
   - Scope: Handles book checkouts, returns, and late fee calculations.
   - Communication Pattern: RESTful API for communication with other services to update book availability, user borrowing history, and user notifications.

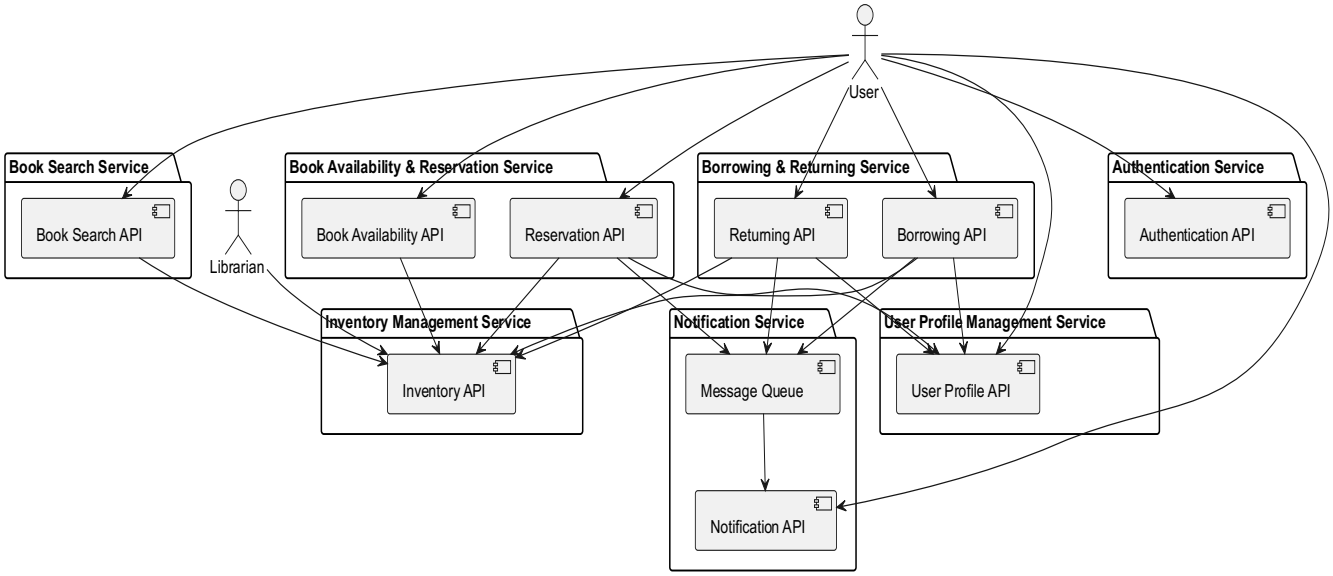
5. User Profile Management Service:
   - Scope: Manages user profiles, borrowing history, and user preferences.
   - Communication Pattern: RESTful API for communication with other services to retrieve and update user information.

6. Notification Service:
   - Scope: Sends notifications for overdue books, reservation updates, and other user-related events.
   - Communication Pattern: Message queues (e.g., RabbitMQ) for receiving messages from other services and sending notifications to users.

7. Inventory Management Service:
   - Scope: Manages the library's inventory, including adding, updating, and removing books.
   - Communication Pattern: RESTful API for communication with other services to update book information and availability.

Architecture diagram that describes the whole system architecture:





Communication pattern used and the limitations around communication for microservices:

In this system, the primary communication patterns for inter-service communication are RESTful APIs for synchronous communication and message queues for asynchronous communication.

For synchronous communication between microservices, RESTful APIs are being used. This form of communication includes sending HTTP requests and getting HTTP responses. Because of their simplicity and ease of use, RESTful APIs are frequently used in a microservice's architecture. However, there are limitations to microservice communication when using RESTful APIs, such as network latency and greater complexity. Because microservices communicate across a network, there is inherent latency in the communication process. When contrasted to monolithic apps, this can result in slower response times. Communication between microservices can sometimes be difficult to implement, especially when dealing with error handling, retries, and timeouts.

Message queues are used to facilitate asynchronous communication among microservices. This communication pattern entails sending messages to and receiving messages from a message queue. Message queues can be used to decouple services and allow for asynchronous processing. However, when employing message queues for communication, there are several restrictions, such as data consistency and fault tolerance. Because each service may have its own data store, ensuring data consistency across several microservices can be difficult. This can eventually lead to inconsistency difficulties. Microservices must also be designed to handle communication failures such as network outages or unresponsive services. This necessitates the use of fault-tolerant communication structures such as circuit breakers and fallback systems.