Group 4

Name : Mary Antonnette V. Bariquit

Alexandria L. Pagalan

Naomi Shene Haclad

Mika P. Yape

Cherry Mae B. Torrepalma

1. Sofware Architectural Styles with its disadvantage and advantage

|  |  |  |
| --- | --- | --- |
| **Architecture Style** | **Advantages** | **Disadvantages** |
| **Layered** | - Easy to maintain and test - Easy to add new layers and data - Easy to divide logic and data | - Performance overhead due to multiple layers - Difficult to optimize with too many layers |
| **Client-Server** | - Centralized control - Easy to manage and scale - Separation of client and server logic | - Network dependency - Server overload with too many requests |
| **Event-Driven** | - Highly scalable and flexible - Loosely coupled components - Reactive system (real-time) | - Complex implementation - Difficult to trace event flow |
| **Microservices** | - Independent deployment - Easier to scale each service - Resilient to failure | - Complex service communication - Overhead in managing multiple services |
| **Repository Style** | - Centralized data management - Consistent data handling - Easy to maintain | - Single point of failure - Bottleneck under high load |

1. Select and existing Software System

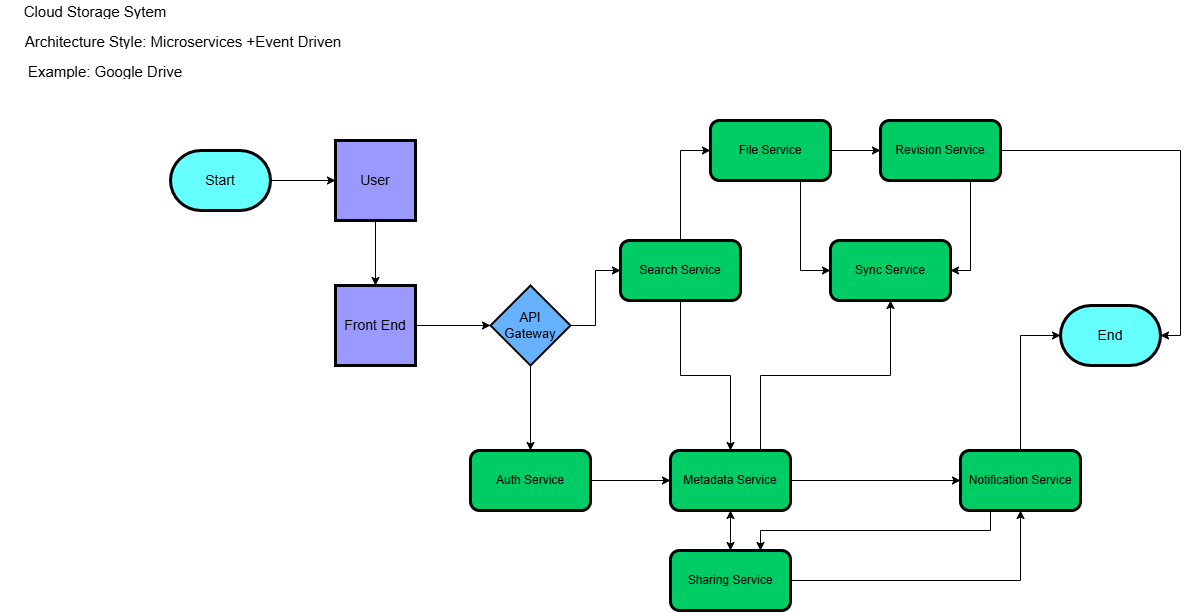
Chosen System:

-Cloud Storage System. (Google Drive)

1. Identify and describe the architecture style it follows.

* Architecture style: Microservices and Event-driven
* Description:
  + - * Cloud Storages System uses a combination of **Microservices** and **Event-Driven Architecture** to manage file storage, access, synchronization, and sharing. Each service handles a specific task independently and communicates with others through a message queue.

1. Basic Block Diagram:



1. Software application for daily use

Chosen Application:

Messenger

1. Identifiy its key components and the relationships

Key Components:

**User**

* The end user who sends and receives messages.
* Can initiate actions like sending messages, making calls, and sharing media.

**Frontend**

* User interface (UI) – Mobile app, web app, or desktop client.
* Handles user inputs (e.g., typing messages, starting calls) and displays real-time updates.

**API Gateway**

* Acts as the central entry point for all backend services.
* Handles routing, load balancing, authentication, and communication between frontend and backend.

**Authentication Service**

* Manages user login and identity verification.
* Ensures secure communication through token-based or OAuth authentication.

**Messaging Service**

* Core component for real-time message exchange.
* Queues and delivers messages between users.

**Media Service**

* Handles sending and receiving of multimedia files (images, videos, audio).
* Compresses and stores media files in cloud storage.

**Notification Service**

* Sends real-time notifications (e.g., message received, call missed).
* Works with push notification systems on iOS, Android, and web.

**Presence Service**

* Tracks user online/offline status.
* Notifies other users when someone becomes available.

**Call Service**

* Manages voice and video calls.
* Handles network connections, encryption, and call quality adjustments.

**Search Service**

* Allows users to search for previous messages, contacts, and files.
* Works with indexing to enable fast searching.

**Group Management Service**

* Manages creation, updates, and permissions for group chats.
* Handles adding/removing members and sending group notifications.

**Sync Service**

* Ensures that conversations are updated across multiple devices.
* Synchronizes message states (read, delivered) and user presence.

**Storage Service**

* Stores messages, media, and user settings.
* Ensures secure storage and fast retrieval.

**Relationships & Flow:**

* **User** - interacts with **Frontend**
* **Frontend** - sends requests to **API Gateway**
* **API Gateway** - routes to **Messaging Service**, **Media Service**, **Call Service**, etc.
* **Sync Service** - ensures that updates appear across devices
* **Notification Service** - sends real-time updates to the user
* **Storage Service** - stores chat history and media files
* **Presence Service** - updates online status

1. Create a UML component diagram structure:



1. Design a simple library management system that allows users to borrow books, return books and search for available books.
2. Identify the functional and non-functional requirements of the system

**Functional Requirements** (What the system should do)

* User Login – The system should authenticate users before granting access.
* Search Book – Users should be able to search for books using keywords or filters.
* Display Results – The system should display search results based on user input.
* Book Details – Users should be able to view details of a selected book.
* Check Availability – The system should determine if the book is available for borrowing.
* Borrow Book – If available, users should be able to borrow the book.
* Return Book – Users should be able to return borrowed books.
* Librarian Claims Book – Librarian should be able to receive returned books.
* Confirm Book Return – Librarian should confirm the return of books.
* Update Status – The system should update the status of books after borrowing or returning.
* Exit or Search Again – Users should be able to either search again or exit the system.

**Non-Functional Requirements** (How the system should perform)

* Performance – The system should process user requests within 2 seconds.
* Availability – The system should be available 99.9% of the time.
* Scalability – The system should handle up to 1,000 concurrent users.
* Security – User data should be encrypted, and only authenticated users can access the system.
* Usability – The interface should be simple and easy to navigate.
* Maintainability – Codebase should be well-documented and easy to update.
* Compatibility – The system should work across web and mobile platforms.
* Reliability – The system should prevent data loss even in case of a failure.

1. Breakdown of the system into subsystems

**User Management Subsystem**

* Handles user authentication and authorization.
* Manages user roles (e.g., librarian, regular user).
* Maintains user profiles and access permissions.

**Search and Retrieval Subsystem**

* Handles search queries for books.
* Retrieves and filters book data based on user input.
* Displays search results in an organized format.

**Book Management Subsystem**

* Stores and manages book information (title, author, genre, availability).
* Handles adding, updating, and deleting book records.
* Manages book borrowing and return status.

**Borrow and Return Subsystem**

* Handles borrowing requests.
* Updates book availability status.
* Manages overdue returns and fines (if applicable).

**Librarian Subsystem**

* Allows librarians to receive returned books.
* Confirms book returns and updates book status.
* Provides an interface for managing book records.

**Notification Subsystem**

* Sends confirmation messages upon borrowing or returning books.
* Alerts users about overdue books.
* Notifies users about book availability.

**Reporting and Analytics Subsystem**

* Generates reports on borrowing trends.
* Provides data on book availability and user activity.
* Monitors system performance and usage.

**Security and Access Control Subsystem**

* Ensures secure login and session management.
* Encrypts sensitive data.
* Manages access based on user roles.

1. Define Parameters such as max books borrowed per user, database scalability needs and performance considerations.

Define Parameters:

**Defined Parameters and System Constraints:**

* **Maximum Books Borrowed Per User**
  + Each user can borrow a maximum of **2 books** at a time.
* **Borrowing Period**
  + The borrowing period is set to **2 weeks (14 days)** from the date of borrowing.
* **Overdue Penalty**
  + A penalty of **10 pesos** will be charged **per day** for each overdue book.

**Database Scalability Needs:**

* The database must handle **thousands of books and users** efficiently.
* Ensure the database is optimized for **fast search and retrieval** of records.
* The system should update the inventory **in real-time** when books are borrowed or returned.
* The database design should support **future scalability** to accommodate growth in the number of books and users.
* Use **indexing** and **optimized queries** to improve search and update performance.
* Regularly perform **database backups** to prevent data loss.
* Ensure **data consistency** and **integrity** during concurrent access (e.g., when multiple users are borrowing or returning books simultaneously).

**Performance Considerations:**

* Implement **caching** where applicable to reduce database load and improve response time.
* Optimize the system to handle **high user traffic** without performance degradation.
* Use **connection pooling** to manage database connections efficiently.
* Minimize **data redundancy** to enhance system speed and accuracy.
* Monitor system performance regularly and apply necessary updates or fixes.

1. Compare different database models (Relational vs NoSQL) and discuss trade-offs.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Relational Database** | **NoSQL Database** |
| **Structure** | Data is stored in tables with predefined schema (rows and columns). | Data is stored in key-value pairs, documents, wide-column stores, or graphs (flexible schema). |
| **Examples** | MySQL, PostgreSQL, SQLite, Microsoft SQL Server | MongoDB, Cassandra, Redis, Amazon DynamoDB |
| **Data Consistency** | Strong consistency with ACID (Atomicity, Consistency, Isolation, Durability) compliance. | Eventual consistency (in most cases) – prioritizes availability and partition tolerance. |
| **Scalability** | Vertical scaling (adding more CPU, memory). | Horizontal scaling (adding more servers). |
| **Schema** | Fixed schema – changes require schema modification. | Flexible schema – allows easy changes in data structure. |
| **Performance** | Slower for high volumes of data due to constraints and joins. | High write and read speed – ideal for handling large volumes of data and real-time processing. |
| **Complex Queries** | Strong support for JOINs and complex queries using SQL. | Limited support for JOINs and complex queries. |
| **Transaction Support** | Strong transaction support. | Limited transaction support (depends on database type). |
| **Flexibility** | Less flexible – data structure needs to be predefined. | High flexibility – dynamic and evolving data structures. |
| **Use Cases** | Financial systems, e-commerce, inventory management, structured data. | Big data, real-time analytics, social media, unstructured data. |

1. Justify the chosen database model on scalability and maintainability

**Relational Database (SQL) Justification**

**Scalability:**

* Supports **vertical scaling** by increasing the server’s capacity (CPU, memory).
* Can handle moderate horizontal scaling through **sharding** and **replication** (but more complex than NoSQL).
* Best for handling structured data with defined relationships.
* Ensures **data consistency** and **integrity** across multiple transactions.

**Maintainability:**

* Provides a **structured schema** that ensures consistency in data storage.
* **ACID compliance** (Atomicity, Consistency, Isolation, Durability) guarantees data integrity.
* Mature tools available for data migration, backups, and schema updates.
* Strong support for handling **complex queries** using SQL.
* Suitable for systems requiring strong consistency and integrity (e.g., a library system).

**NoSQL Database Justification**

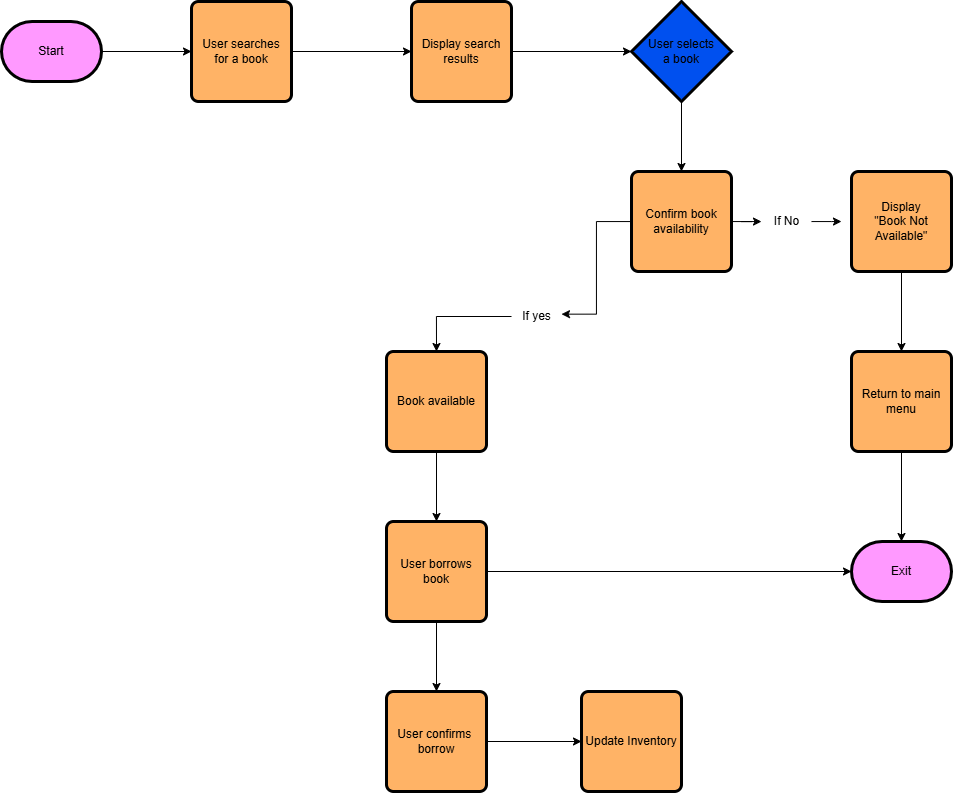
**Scalability:**

* Supports **horizontal scaling** by distributing data across multiple servers.
* Can handle large-scale, high-traffic workloads with fast data retrieval.
* Ideal for handling **massive volumes of unstructured or semi-structured data**.
* Ensures high availability and fault tolerance through replication.

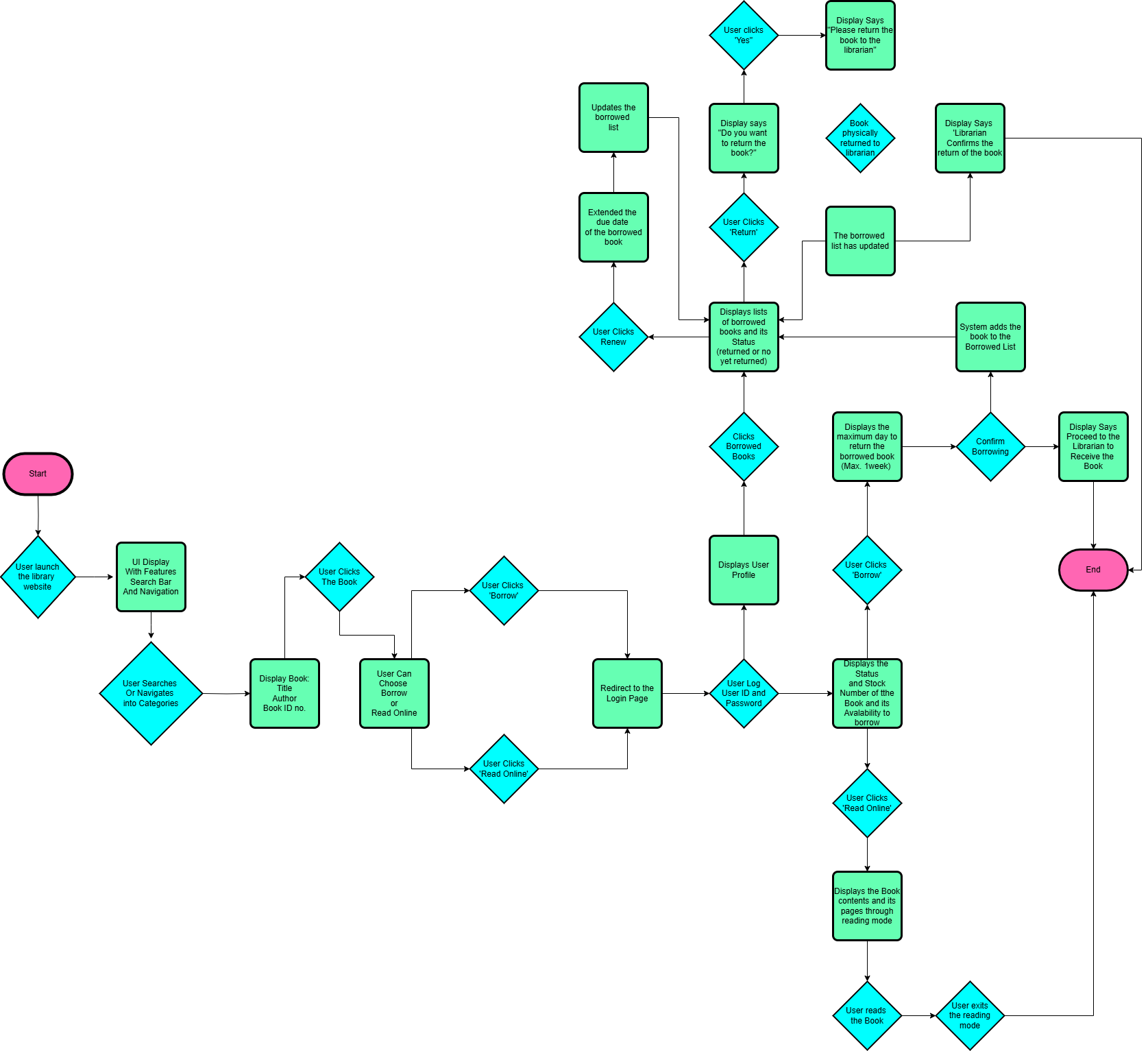
**Maintainability:**

* Schema-less structure allows for flexible and dynamic data models.
* Better suited for **evolving or rapidly changing data**.
* Suitable for storing large amounts of varied data (e.g., logs, user behavior).
* Provides flexibility in handling different types of data without a fixed schema.

1. Sketch a high level prototype UI for book search and borrowing.



1. Create the following UML diagrams for the library management system:



1. Identify key actors and interactions

**Key Actors and Interactions**

**Primary Actors:**

* **User** – The person who searches for, borrows, and returns books.
* **Librarian** – The person who verifies book returns and manages the inventory.

**Interactions:**

* **User Interactions:**
  + Launches the library system
  + Searches for a book
  + Views search results and selects a book
  + Borrows a book (if available)
  + Confirms the borrow request
  + Returns a book
  + Renews a borrowed book
  + Exits the system
* **Librarian Interactions:**
  + Confirms book return
  + Updates the inventory
  + Manages the borrowing list

1. Define classes such as User, Book, Loan, etc.

**Classes Definition**

Here are the key classes for the library management system:

* **User Class**
  + **Attributes:**
    - user\_id – Unique identifier for each user
    - name – User’s full name
    - email – User’s email address
    - borrowed\_books – List of books borrowed by the user
  + **Methods:**
    - search\_book(title) – Search for a book by title
    - borrow\_book(book) – Borrow a book
    - return\_book(book) – Return a borrowed book
    - renew\_book(book) – Renew the borrowing period of a book
* **Book Class**
  + **Attributes:**
    - book\_id – Unique identifier for each book
    - title – Title of the book
    - author – Author of the book
    - status – Availability status of the book (Available, Borrowed)
    - due\_date – Due date for borrowed books
  + **Methods:**
    - get\_book\_info() – Get details about the book
    - update\_status(status) – Update availability status
* **Loan Class**
  + **Attributes:**
    - loan\_id – Unique identifier for each loan
    - user\_id – ID of the user who borrowed the book
    - book\_id – ID of the borrowed book
    - borrow\_date – Date the book was borrowed
    - due\_date – Date the book is due
  + **Methods:**
    - create\_loan(user, book) – Create a new loan
    - calculate\_penalty() – Calculate the penalty for overdue books
    - return\_book() – Mark a loan as returned
* **Librarian Class**
  + **Attributes:**
    - librarian\_id – Unique identifier for each librarian
    - name – Librarian’s name
  + **Methods:**
    - confirm\_return(book) – Confirm the return of a book
    - update\_inventory(book) – Update the status of a book in the inventory
* **Inventory Class**
  + **Attributes:**
    - books – List of all books in the library
  + **Methods:**
    - add\_book(book) – Add a new book to the inventory
    - remove\_book(book) – Remove a book from the inventory
    - search\_book(title) – Search for a book in the inventory
    - update\_book\_status(book, status) – Update the status of a book

1. Illustrate a book borrowing process.

* **User** sends a search request (searchBook(title)) to the **Library System**.
* **Library System** checks the availability of the book in the **Inventory**.
* **Inventory** returns the availability status to the **Library System**.
* If available, **User** selects the book and sends a borrow request (borrowBook).
* **Library System** updates the status of the book in the **Inventory** to Borrowed.
* **Library System** creates a new **Loan** record.
* **Library System** confirms the successful borrowing to the **User**.

**Conclusion**

The library management system is designed to streamline the process of searching for, borrowing, and returning books efficiently. By breaking down the system into functional components, the design ensures that user interactions are straightforward while maintaining system performance and security. The defined classes such as User, Book, Loan, and Librarian effectively capture the key entities involved in the process. The borrowing sequence is managed through clear interactions between the user interface, backend processing, and database updates. This structured design enables scalability, consistency, and ease of maintenance, ensuring that the system can handle large user loads and future growth.

**Key Takeaways**

* **Structured Design:** The library management system follows a structured, layered design that separates key components for better performance and maintenance.
* **Efficient User Experience:** The system provides an intuitive interface for searching, borrowing, and returning books.
* **Scalability and Performance:** The database and system design enable handling of high user traffic and large volumes of data without performance degradation.
* **Security and Integrity:** User authentication, secure data handling, and consistent system updates ensure data protection and reliability.
* **Flexible and Maintainable:** The system's modular design allows for easy updates and future expansion.
* **Role Management:** Clear distinction between user and librarian roles ensures proper access control and accurate inventory management.
* **Automation and Real-Time Updates:** Automated status updates and notification handling improve user satisfaction and reduce operational overhead.