TEAM 1: Dalangin, Vea

Bosito, Jhon Aldrin

Codilla , Christian

I. Review different software architectural styles:

Architecture Style	Advantages	Disadvantages
Layered Architecture	- Separation of concerns - Easy to maintain - Reusability	- Can cause performance bottlenecks - Difficult to scale
Client-Server	- Easy to implement - Clear separation of client and server - Scalable	- Centralized failure point - Network latency can be an issue
Event-Driven	- Highly scalable - Separation of components - Flexible	- Complexity in event management - Debugging and tracing is harder
Microservices	- Scalability and flexibility - Independent deployability - Failure isolation	- Complex communication - Overhead in managing multiple services
Repository Style	- Centralized data - Reusable components - Easy data access	- Can lead to performance bottlenecks - Complex data access logic

II. Select an Existing Software System: Facebook

Architecture Style: Client-Server with Microservices

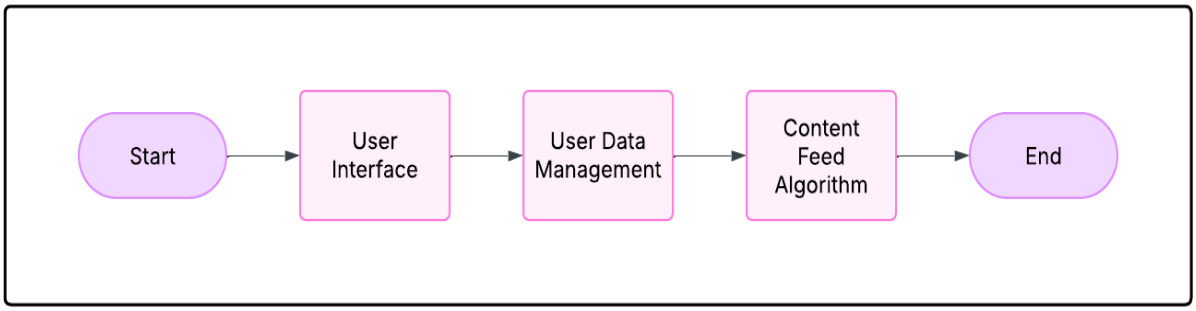
Facebook uses a Client-Server model, where your device (client) communicates with its servers. The backend is divided into microservices, each handling specific tasks like News Feed, Messenger, and Ads. This setup helps Facebook scale, update, and maintain its system efficiently while ensuring reliability and performance.

Facebook follows a microservices architecture combined with a client-server model:

- Client-Server: Your device (client) communicates with Facebook's servers to load content or perform actions.

- Microservices: The backend is split into small, independent services (News Feed, Messenger, and Ads) that handle specific tasks. This allows Facebook to scale, update, and maintain its system efficiently while ensuring high performance and reliability.

Basic block diagram:



III. Choose a Software Application You Use Daily: Spotify

Key Components of Spotify:

1. **User Interface (UI):**

- The front-end through which users interact with Spotify. It displays features like playlists, music search, and recommendations.
- **Components:**
    - Music Player: Plays audio tracks.
    - Search Functionality: Allows users to search for songs, artists, and albums.
    - Playlist Management: Manages user-created playlists, favorites, and recommendations.

2. **Authentication Service:**

- Handles user sign-in and authentication, ensuring that only registered users can access their profiles and playlists.

3. **Recommendation Engine:**

- Analyzes user data and preferences (listening history, likes) to provide personalized music recommendations.

4. **Music Streaming Service:**
   Streams audio content to users, delivering songs, albums, and podcasts.
- Handles media requests and streaming quality adjustments based on the user's network.

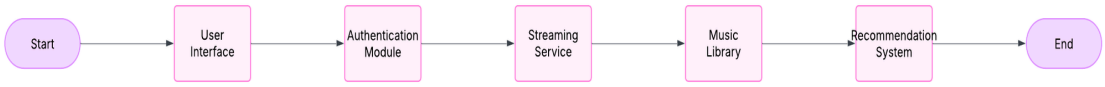
5. **Database:**
- Stores user data, playlists, music metadata (songs, albums, artists), and listening history.
- Relational Database for structured data (user profiles) and NoSQL for unstructured data (music metadata).

6. **Notification Service:**
   Sends notifications for new music releases, followers, and playlist updates.
7. **Payment Service:**
- Manages user subscriptions, payments, and premium accounts.

Relationships:

- **UI** interacts with **Authentication Service** for user login.
- **UI** communicates with **Music Streaming Service** to play tracks.
- **UI** accesses the **Recommendation Engine** for music suggestions based on user preferences.
- **Music Streaming Service** retrieves data from the **Database** to stream music.
- **Authentication Service** and **Payment Service** are connected to ensure valid user accounts and manage subscriptions.
- **Database** stores both user data and music data, which is accessed by multiple services.

UML component diagram:



IV. Library Management System Design

To design a simple Library Management System (LMS) that allows users to borrow books, return books, and search for available books, let's break down the requirements, identify subsystems, and define the necessary parameters.

Functional Requirements

1. **User Management**
   - **User Registration:** Users should be able to register an account in the system.
   - **Login/Logout:** Users should be able to log in and log out of the system.
   - **User Profiles:** Store information about users (name, email, membership ID, borrowed books, etc.)
2. **Book Inventory**
   - **Add Books:** Admins can add new books to the system with details such as title, author, ISBN, genre, and availability.
   - **View Available Books:** Users can search and view a list of available books.
   - **Book Details:** Users can view details of a book, including a description, author information, etc.

3. **Borrowing System**
    - **Borrow Book:** Users can borrow books if available, with a limit on the number of books they can borrow.
    - **Return Book:** Users can return borrowed books.
    - **Due Date & Late Fees:** Track due dates for borrowed books and charge late fees if applicable.
4. **Search Functionality**
    - **Search by Title:** Users can search books by title.
    - **Search by Author:** Users can search books by author.
    - **Search by Genre:** Users can filter books based on genre or category.
5. **Admin Panel**

   View Borrowing History: Admins can see a history of borrowed books and user activities.
    - **Manage Book Inventory:** Admins can update, remove, or modify book details in the system.
    - **Manage Users:** Admins can view and manage user accounts, such as suspending accounts for violations.

Non-Functional Requirements

1. **Scalability**
    - The system should be able to scale as the number of users and books increases. The database design should allow for efficient handling of large datasets (books, users, borrow records).
2. **Availability**
    - The system should be highly available and provide consistent access to users, even during high load times.
3. **Performance**
    - The system should provide quick search results for users.
    - The borrowing process should be fast and efficient.
4. **Security**

   Sensitive information (user profiles, borrowing history) must be securely stored (e.g., encrypted passwords, secure login sessions).
    - Ensure that unauthorized users cannot access or modify the system.
5. **Backup & Recovery**

   Regular backups of the system data (books, users, borrowing history) should be taken to prevent data loss.
    - The system should have a disaster recovery plan in place.
6. **Usability**

   The system should be easy to use, with an intuitive user interface for both the users and admins.
    - It should support multiple device types (desktop, mobile).
7. **Compliance**
    - The system must comply with data protection and privacy laws (GDPR).

Key Parameters and Design Considerations

1. **Max Books Borrowed per User**
    - **Parameter:** Set a limit, 5 books per user at a time.
    - **Design Consideration:** This can be enforced through the Borrowing Service, which checks the number of books a user has before allowing further borrowing.
2. **Database Scalability**
    - **Parameter:** The system must support a growing number of users and books.
    - **Design Consideration:** Use a relational database (PostgreSQL or MySQL) with optimized indexing on frequently searched fields (ISBN, title, author). In case of growth, consider horizontal scaling or sharding the database.
3. **Performance Considerations**
    - **Parameter:** Quick search and borrowing operations.
    - **Design Consideration:** Implement caching for frequent searches, such as a list of most popular books or genres. Use indexes on search fields like title, author, and genre.
4. **Late Fees**
    - **Parameter:** Define the late fee, $1 per day after the due date.
    - **Design Consideration:** Store due dates in the database and run periodic checks to calculate overdue charges for users.
5. **Backup and Recovery**
    - **Parameter:** Backup frequency (daily backups).
    - **Design Consideration:** Implement automated backup routines for both the database and application data. Use cloud-based storage (AWS S3) for easy recovery.
6. **Security**
    - **Parameter: Im**plement strong user authentication and data protection (password hashing, session tokens).
    - **Design Consideration:** Use libraries like “bcrypt” for password hashing and JWT (JSON Web Tokens) for secure session management.

Comparison of Database Models: Relational vs NoSQL

1. Relational Databases (RDBMS)

Relational databases (MySQL, PostgreSQL, SQL Server) store data in structured tables with predefined schemas. These databases are designed to handle structured data and relationships between entities.

- **Structure**: Data is organized in tables with rows and columns.
- **Schema**: The schema is rigid, meaning the structure of the data (tables, fields, relationships) must be defined upfront.
- **ACID Compliance**: RDBMS guarantees ACID properties (Atomicity, Consistency, Isolation, Durability), ensuring data integrity.
- **Scalability**: Scaling relational databases vertically (increasing server capacity) is generally easier, but horizontal scaling (distributing data across multiple servers) can be complex.
- **Use Case**: Best suited for applications with structured data and complex queries requiring joins and transactions (banking systems, ERP applications).

2. NoSQL Databases

NoSQL databases (such as MongoDB, Cassandra, CouchDB) are designed to handle unstructured or semi-structured data. They support flexible schemas, horizontal scaling, and are optimized for specific use cases.

- **Structure**: Data can be stored in various formats, such as key-value pairs, documents, columns, or graphs.
- **Schema**: NoSQL databases are schema-less, allowing for flexible data structures that can evolve over time.
- **ACID Compliance**: Many NoSQL databases offer eventual consistency rather than strict ACID compliance. This is often fine for distributed systems that prioritize availability and partition tolerance (according to the CAP theorem).
- **Scalability**: NoSQL databases are designed to scale horizontally, meaning they can handle high volumes of data by distributing the load across multiple servers.
- **Use Case**: Best suited for applications with large amounts of unstructured or semi-structured data, high transaction volumes, or needing real-time data processing (social media, big data analytics, IoT).

Trade-offs of NoSQL Databases:

- **Pros:**
    - Highly scalable and flexible in handling various data models (JSON, key-value, graph).
    - Easier to scale horizontally (across multiple servers).
    - Can efficiently handle high volumes of unstructured or semi-structured data.
- **Cons:**
    - Lack of strong consistency in some cases (eventual consistency model).
    - Less mature than RDBMS for complex queries.
    - No standardized query language (though some NoSQL databases provide their own query languages).

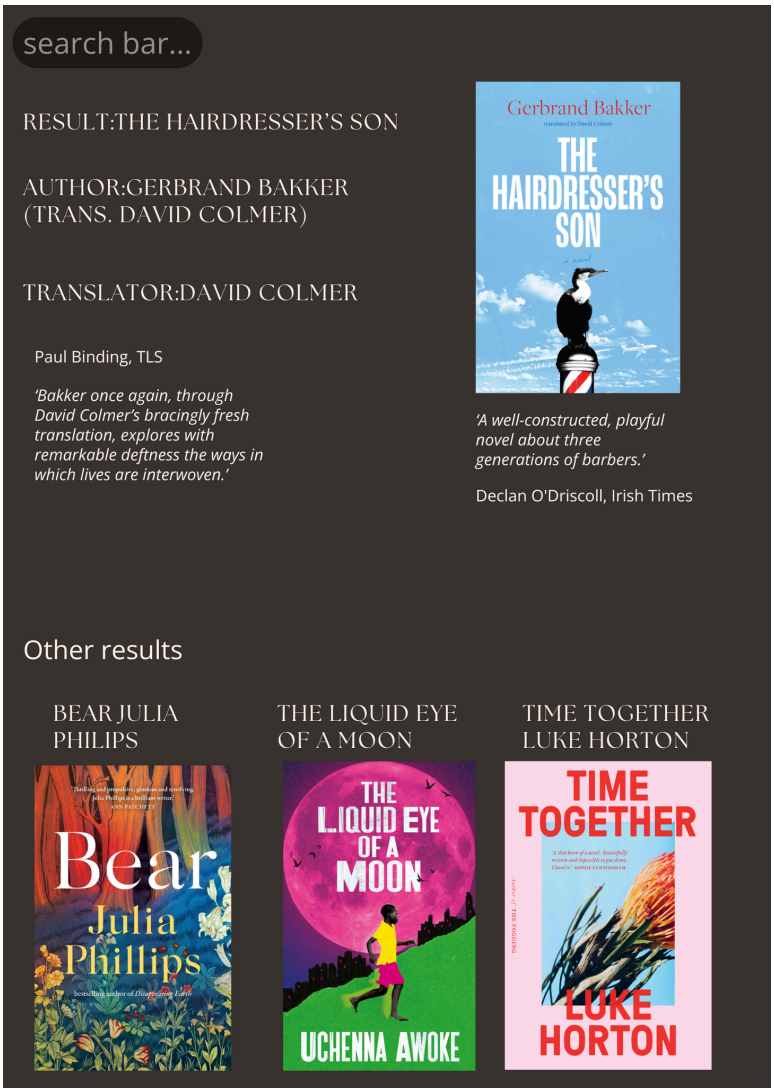
Justifying the Choice Based on Scalability and Maintainability:

Library Management System:

1. **Scalability:**
    - **NoSQL Database:** A NoSQL database like MongoDB or Cassandra would be more suitable for high scalability. For example, as the system grows and needs to accommodate a growing number of books, users, and potentially millions of transactions, NoSQL would provide better horizontal scalability, making it easier to distribute data across multiple servers.
2. **Maintainability:**

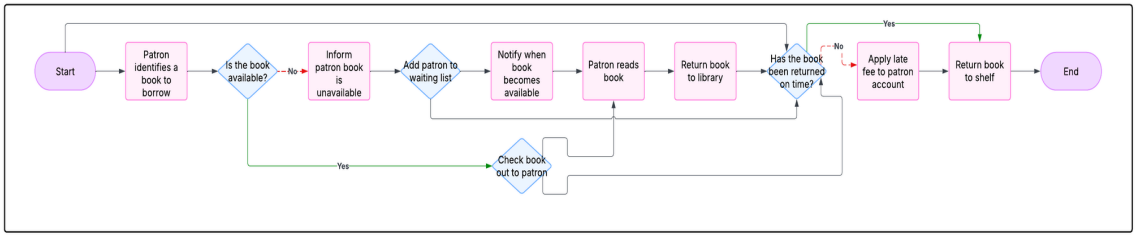
    - **NoSQL Database:** If the system might require frequent schema changes (such as changes in metadata for books, user preferences, etc.), NoSQL databases allow for more flexible schema evolution over time, making it easier to maintain and adapt to changing requirements.

Sketch a high-level prototype UI for book search and borrowing:

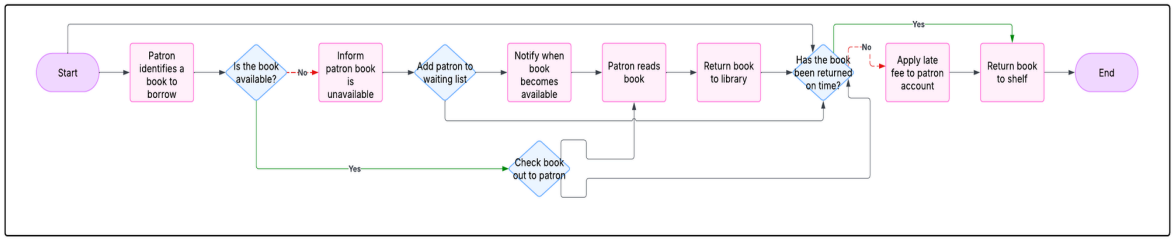


UML diagrams for the library management system:

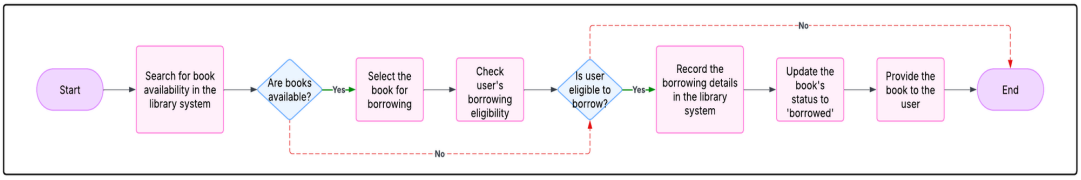
: Identify key actors and interactions.



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



: Illustrate a book borrowing process.



: Show system components and their interactions.