Instructions to start the Library Application

1. Install flask by running this command: `python pip install flask`
2. Generate the database by running: `python database.py`
3. Run the application by running: `python app.py`
4. In the terminal, there will be a http address; hold ctrl then click it to view the application in your browser.

Step (2): Project Specifications

1. **Items in the library**
    - **Types**: Print books, online books, magazines, scientific journals, CDs, records, and maybe some other media.
    - **Attributes**: Each item has a title, author or creator, publication date, genre, and type. Items may have multiple copies, each with a condition (e.g., new, good, worn) and availability status.
    - **Management**: Items are tracked for borrowing and also for event recommendations.
2. **Users (Library members)**
    - **Role**: People who borrow items and attend events. Users can also be a volunteer. Users can also request help if needed.
    - **Attributes**: Name, contact information (e.g., email or phone), unique membership ID, total fines owed from borrowing a copy of an item, and a binary flag indicating if they volunteer.
    - **Activities**: Borrow items, return them, pay fines, attend events, volunteer, or request help.
3. **Borrowing system**
    - **Process**: Users borrow specific copies of items with a borrow date and due date, and when they return that copy, the return date will be recorded.
    - **Fines**: Late returns incur fines (we can set the fine at $0.50 per day), accumulated in the user's total fines.
    - **Tracking**: Availability of copies is updated upon borrowing and returning.
4. **Events**
    - **Types**: Book clubs, art shows, film screenings, and other library-hosted activities.
    - **Attributes**: Event name, date, time, description, recommended audience (e.g., adults, kids), and the social room where it's held.
    - **Access**: Free attendance, with users registering to participate.
    - **Recommendations**: Certain items (e.g., books) may be recommended for events.
5. **Social rooms**
    - **Purpose**: Venues for the events will be the rooms of the library.
    - **Attributes**: Room name and capacity (number of people it can hold).
6. **Personnel**
    - **Purpose**: It's a subclass of Users
    - **Role**: Librarians, assistants, and other staff managing items and events.
    - **Attributes**: Position (inherits Users attributes)
7. **Future items**
    - **Purpose**: Items planned for future addition to the library collection.

- **Attributes**: Title, author, type, and expected arrival date.

8. **Additional features**
    - **Volunteering**: Users can volunteer for the library. We can track it via a yes/no flag.
    - **Help Requests**: Users can request assistance from a personnel, with details like request date, issue, and status (e.g., pending, resolved).

Relationships

- Users borrow copies of items.
- Users attend events.
- Personnel manage items and events.
- Events are held in social rooms (one room per event, multiple events per room over time).
- Items are recommended for events.
- Users request help from personnel.

Based on the specification, we can make a table, that will look like the one below,

Table Name	**Attributes**	**Foreign Key (FK)** `` `ColumnName → ReferencedTable(ReferencedColumn)` ``
Item	ItemID (PK), Title, Type, Author, PublicationDate, Genre	
Copy	CopyID (PK), ItemID (FK), Condition, Availability	ItemID → Item(ItemID)
User	UserID (PK), Name, ContactInfo, MembershipID, TotalFines, IsVolunteer	
SocialRoom	RoomID (PK), Name, Capacity	
Event	EventID (PK), Name, Date, Time, Description, RecommendedAudience, RoomID (FK)	RoomID → SocialRoom(RoomID)
Personnel	PersonnelID (PK), Name, Position, ContactInfo	
FutureItem	FutureItemID (PK), Title, Author, Type, ExpectedArrivalDate	
Borrows	UserID (FK, PK), CopyID (FK, PK), BorrowDate (PK), DueDate, ReturnDate	UserID → User(UserID), CopyID → Copy(CopyID)
Attends	UserID (FK, PK), EventID (FK, PK)	UserID → User(UserID), EventID → Event(EventID)

Table Name	Attributes	Foreign Key (FK) `ColumnName → ReferencedTable(ReferencedColumn)`
ManagesEvent	PersonnelID (FK, PK), EventID (FK, PK)	PersonnelID → Personnel(PersonnelID), EventID → Event(EventID)
ManagesItem	PersonnelID (FK, PK), ItemID (FK, PK)	PersonnelID → Personnel(PersonnelID), ItemID → Item(ItemID)
RecommendedFor	ItemID (FK, PK), EventID (FK, PK)	ItemID → Item(ItemID), EventID → Event(EventID)
HelpRequest	RequestID (PK), UserID (FK), PersonnelID (FK), RequestDate, Issue, Status	UserID → User(UserID), PersonnelID → Personnel(PersonnelID)

Step (3): E/R Diagrams

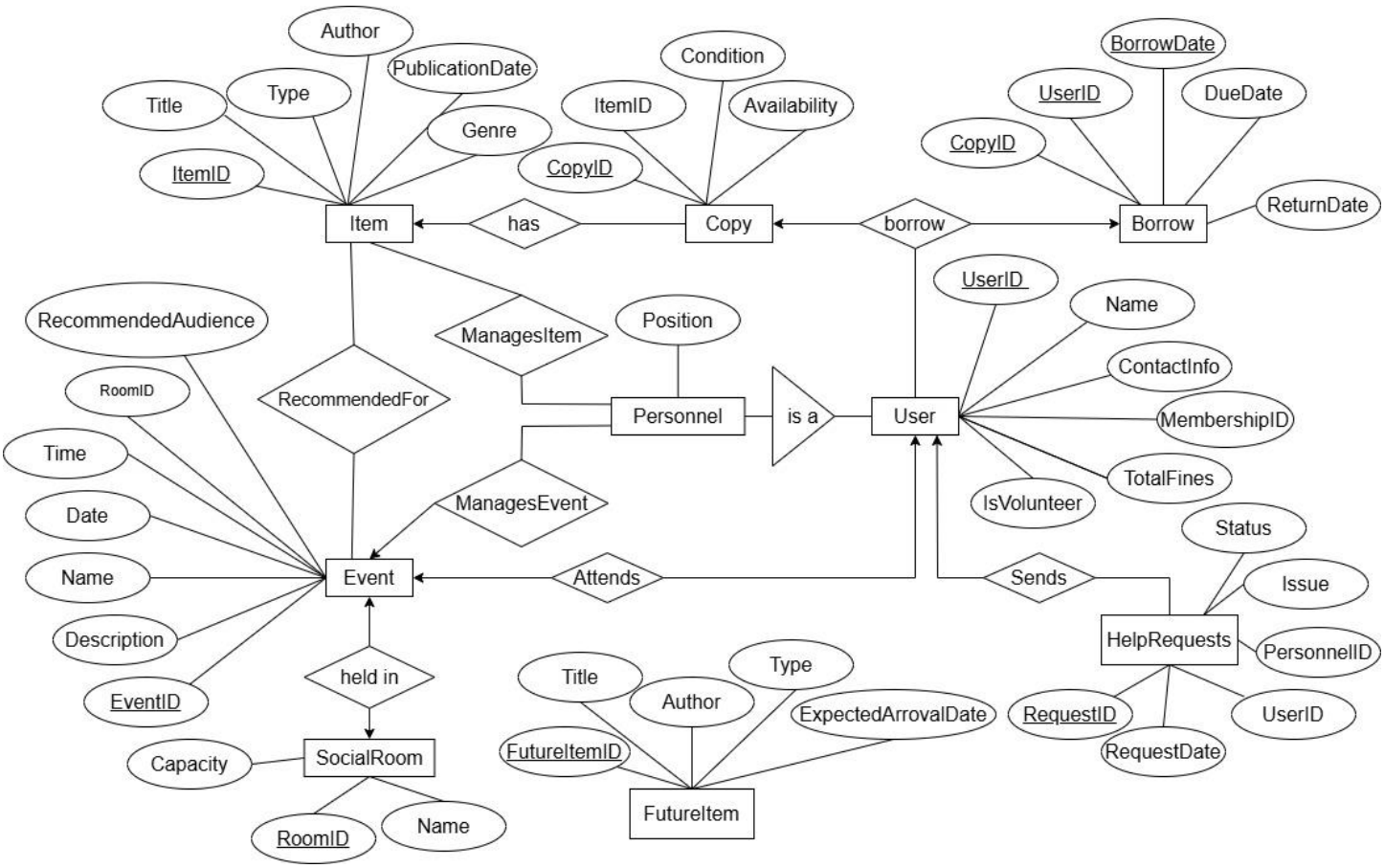

Figure 1: ERD for Library DB(draw.io link)

Step (4): Does the Library design allow anomalies?

To check for anomalies, first, we converted the ER diagram into relational tables. Then, we analyze functional dependencies (FDs) to ensure they are in Boyce-Codd Normal Form (BCNF).

A relation with FDs 'F' is in BCNF if for every X → Y in 'F'

- Y ⊆ X (the FD is trivial), or
- X is a key

We will analyze each table below,

Table Name	Details
Item	FD: ItemID → Title, Type, Author, PublicationDate, Genre. ItemID is the PK. BCNF: Yes
Copy	FD: CopyID → ItemID, Condition, Availability. CopyID is the PK. BCNF: Yes
User	FDs: UserID → Name, ContactInfo, MembershipID, TotalFines, IsVolunteer. MembershipID → Name, ContactInfo, UserID, TotalFines, IsVolunteer. UserID is the PK, and all attributes depend on it. MembershipID is an alternate key, but UserID as PK satisfies BCNF. Both FDs satisfy BCNF. BCNF: Yes
SocialRoom	FD: RoomID → Name, Capacity. RoomID is the PK. BCNF: Yes
Event	FD: EventID → Name, Date, Time, Description, RecommendedAudience, RoomID. EventID is the PK. BCNF: Yes
Personnel	FD: PersonnelID → Name, Position, ContactInfo. PersonnelID is the PK. BCNF: Yes
FutureItem	FD: FutureItemID → Title, Author, Type, ExpectedArrivalDate. FutureItemID is the PK. BCNF: Yes
Borrows	FD: (UserID, CopyID, BorrowDate) → DueDate, ReturnDate. Composite PK determines all attributes. BCNF: Yes
Attends	FD: (UserID, EventID) → (no additional attributes). Composite PK, no FDs beyond the key. BCNF: Yes
ManagesEvent	FD: (PersonnelID, EventID) → (no additional attributes). Composite PK. BCNF: Yes

Table Name	Details
ManagesItem	FD: (PersonnelID, ItemID) → (no additional attributes). Composite PK. BCNF: Yes
RecommendedFor	FD: (ItemID, EventID) → (no additional attributes). Composite PK. BCNF: Yes
HelpRequest	FD: RequestID → UserID, PersonnelID, RequestDate, Issue, Status. RequestID is the PK. BCNF: Yes

Result

All tables are in BCNF, as every non-trivial FD has a superkey on the left side of the arrow(→). So, no decomposition is needed. Additional constraints (for example: Availability consistency with Borrows, TotalFines is calculated from Borrows) are handled via application logic or triggers, not FDs.

- Availability Consistency with Borrows?

  The **Copy** table has an attribute called **Availability** (1 = available, 0 = not available).

  A Copy should be marked as unavailable (Availability = 0) when it's currently borrowed (i.e., there's a row in the Borrows table with that CopyID and no ReturnDate). When the Copy is returned (ReturnDate is set), Availability should be set back to 1.

  So how will we handle this?

  → **We cannot with FDs**: An FD would look like: CopyID → Availability

  In the copy table, we have: FD: CopyID → ItemID, Condition, Availability. [step 1]

  Decomposition from step 1: CopyID → Availability

  But, there is no FD in the Copy table that says, "If a CopyID is in Borrows with ReturnDate = NULL, then Availability = 0." FDs can't look at another table or check for NULL values dynamically.

  → **We can do this with Application logic:**

  - When a user borrows a Copy (inserts a row into Borrows), your program (e.g., in Python) checks:
    - "Is this Copy available? (Availability = 1)"
    - If yes, set Availability = 0 in the Copy table and insert the Borrows row.
  - When a user returns a Copy (updates ReturnDate in Borrows), the program sets Availability = 1 in the Copy table.

- TotalFines is calculated from Borrows?

  The **User** table has an attribute **TotalFines**, which stores the total fines owed by a User.

  TotalFines should be calculated based on overdue items in the Borrows table:

  For each borrowing (row in Borrows), if the current date is past the DueDate and ReturnDate is NULL, calculate a fine (we can say, $0.50 per day late).

  Sum all fines for that User to get TotalFines.

  So how will we handle this?

  → **We cannot with FDs**: An FD would look like: UserID→TotalFines

  In the user table: UserID → Name, ContactInfo, MembershipID, TotalFines, IsVolunteer. [step 1]

  Decomposition from step 1: UserID→TotalFines

  But, the calculation of TotalFines isn't an FD because: TotalFines isn't determined by a static rule within the User table. It depends on data in Borrows (DueDate, ReturnDate) and the current date, which changes over time.

→ **We can do this with Application logic:**
To find a user's total fines, start with zero. Get today's date. Then, find all the books the user has borrowed. For each book, if it hasn't been returned and it's past the due date, count how many days it's late. Multiply the late days by 0.50 to get the fine(assuming, the fine is $0.50 per day). Add this to the total. After checking all books, update the user's fine in the system.

So, this makes the design free of insertion, deletion, and update anomalies related to FDs.