System Architecture Overview

The system follows a client server architecture. The client is a desktop application that uses Tkinter. The backend server is built with Flask and responds to HTTP requests. The MySQL database, holds all permanent data and communicates with the server. The client does not talk directly with the database. All interactions go through the backend. This blocks unapproved access to information and guarantees that every operation follows the system's regulations. The backend also manages validation, permissions, and data formatting. The client must gather user input and present the outcomes. The complete request cycle appears as follows. Initially, the user presses a button or provides input in the Tkinter interface. The API client class facilitates the creation of an HTTP request by the client. The request is forwarded to the relevant endpoint in the Flask backend. The backend obtains the request, validates the data, executes logic, and processes SQL queries via its data access functions. The database provides results, which the backend formats into a JSON response. The client gets the reply and refreshes the interface. This happens for each action, including signing in, searching, adding products to a cart, or finalizing an order. The client manages the user interface, the backend handles the logic, and the database manages the storage.

Database Design Overview

The database is built using a relational model because the data has clear relationships. The database contains four main tables: users, books, orders, and order_items. Each table represents an essential concept in the system. This design avoids duplicate data and ensures accuracy by using foreign keys to link records. All tables were created with normalization in mind, making the system easier to manage and less likely to contain inconsistencies. The structure avoids duplication. For example, book titles are not copied into orders. Instead, orders only store a link to the books table. This makes updates easier and prevents errors when things change.

Users Table

The users table contains information for all accounts in the system. This includes both customers and managers. Each user has a unique username and email, a password hash, and a role field that determines whether the user is a regular customer or a manager. The password is not stored directly. Instead, a hash is stored to improve system security. Every time a user logs in, the backend checks the hash using a secure algorithm. A user can have multiple orders associated with their account. This creates a one to many relationship between users and orders. The user ID is stored in the orders table as a foreign key. Deleting a user automatically removes their orders and order items because cascading delete rules apply.

Books Table

The book's table holds all book related information. Each row includes the title, author, buy price, rent price, an availability flag, and timestamps. The availability flag is helpful because a manager can hide a book from customers without deleting it. Storing buy and rent prices separately gives the system flexibility in offering different pricing models.
Books can appear in many orders, but the book table itself does not store any direct reference to orders. Instead, the connection is handled through the order_items table. This design prevents duplication. For example, the title of a book is stored once in the books table rather than copied into every order.

Orders Table

The order table records each order created by a customer. Includes user ID, total amount, payment status, and creation timestamp. The payment status starts as Pending. Managers can update it to Paid or Canceled. Since each user can place many orders, the orders table contains a foreign key linking it to the user.

Order Items Table

The order_items table records the details of each item inside an order. This includes the order it belongs to, the book being purchased or rented, the transaction type, the price at the time, and the timestamp. Storing the price at the time of the order prevents issues if a book's price changes later. For example, if a book increases in price, past receipts must still reflect the original amount charged.

This table links orders and books, forming a many to many relationship. A single order can have multiple books. A single book can appear in many different orders. The order_items table ensures that each record connects one order to one book.

Database Design Decisions

The database was designed to keep data clean and easy to manage. By avoiding repeated information, the system stays consistent. Foreign keys were used to maintain integrity. Cascading delete rules were added to automatically remove related items when a user or order is removed.

Indexes were created for columns that are frequently used in searches or filtering. For example, the username and email fields have indexes, so login queries are fast. Book titles and authors also have indexes to speed up search results. Fields like created_at and payment_status have indexes so that sorting and filtering manager reports are as fast as well.

Application Design Overview

The application design focuses heavily on the Flask backend. The backend is responsible for handling all communication between the client and the database. The backend uses a modular structure similar to the model view controller pattern. It includes routes for different parts of the system, model functions for database access, and helper functions for tasks like authentication and sending emails. Authentication routes handle registration and login. Books routes handle searching and listing books. Orders routes handle the customer checkout process. Manager routes handle administrative tasks. This separation makes it easy to navigate the backend code and understand which part controls which feature. The backend uses decorators. These decorators automatically enforce that certain routes require authentication or specific roles. The login_required decorator checks for a valid JWT token in the request header. The role_required decorator ensures that only managers can access particular routes. This prevents unauthorized users from performing actions that they should not be allowed to do. The backend also contains utility modules. One module handles sending emails to customers. Another module caches book results to speed up repeated book searches. The data access layer contains functions that execute SQL queries. This structure centralizes database communication and keeps query logic in one place.

Frontend Design with Tkinter

The frontend is built using Tkinter while designed to be simple and easy to use. The interface includes login, registration, book browsing, cart viewing, order placement, and manager tasks. The Tkinter application makes calls to the backend through an API client class. This class wraps the requests library and automatically attaches the JWT token to requests when needed. Using background threads prevents the interface from freezing when a user performs time-consuming tasks such as loading books or placing an order. Upon logging in, the program verifies the role field provided by the backend.

Implementation Summary

The implementation brings together the database, backend, and frontend. For authentication, the backend checks for duplicate usernames and email addresses, securely hashes passwords, and generates a token when a user logs in. The client stores the token to authenticate future requests. For book searching, the client sends the search text to the backend. The backend performs an SQL query using wildcards to match titles or authors. Results are returned as JSON and displayed in a Tkinter table. For checkout, the client sends the list of books in the cart. The backend validates the books, calculates totals, inserts new records into the orders and order_items tables, and returns the final order details. For manager tasks, the backend performs join queries between orders and users so that managers see customer information alongside order details. Managers can update payment statuses, and the backend performs SQL update queries to reflect these changes. Inventory editing is done through insert and update queries in the books table.