Internal CRUD Project: Transactional LibroAPI (Library Management System)

Project Objective & Use-Case Definition

A local community library is expanding its digital modernization to include not just catalog management, but also **book borrowing and returning**. Their current system struggles with concurrency, leading to inconsistencies where a book might show as "available" when it's not, or a loan record disappears.

Your mission is to build the backend service—the "brain"—for their new digital lending system. You will create a REST API called **LibroAPI** that not only allows librarians to manage their book collection but also reliably handles the core operations of **borrowing** and **returning** books. This API will be the foundation for future applications, like a public-facing search and reservation website or a mobile app for members.

The core challenge is ensuring data integrity for interdependent operations, which you will solve using Database Transactions.

Key Use-Cases:

1. **Book Catalog Management (CRUD):**
    - **Create:** Add a new book to the catalog (including defining its total number of copies).
    - **Read:** Look up a specific book's details or browse the entire collection.
    - **Update:** Correct or add information for an existing book (e.g., increase total copies).
    - **Delete:** Remove a book from the catalog.
2. **User Management:** (Simplified for this project)
    - **Create:** Register a new library member.
    - **Read:** Look up member details.
3. **Book Borrowing:**
    - A library member borrows an available copy of a book. This must reliably:
        - Decrement the `available_copies` for the book.
        - Create a new `Loan` record tracking who borrowed it, when, and when it's due.
    - **Crucially, these two operations must succeed OR fail together.**
4. **Book Returning:**
    - A library member returns a borrowed book. This must reliably:
        - Increment the `available_copies` for the book.
        - Update the existing `Loan` record to mark it as `returned` and record the `return_date`.
    - **Again, these two operations must succeed OR fail together.**
5. **Loan Tracking:**
    - View all active loans.
    - View loans by a specific user or for a specific book.

2. Learning Outcomes

By completing this project, you will not only learn technical skills but also understand how they apply to solving complex real-world problems. You'll gain hands-on experience with:

- **Solving a Business Need:** Translating business requirements (digitizing a catalog, managing loans reliably) into a robust technical solution (a REST API with transactional integrity).
- **API Development:** Creating a fully functional, use-case-driven REST API with FastAPI.
- **Production Database Interaction:** Modeling and managing multiple related data entities in a PostgreSQL database using SQLAlchemy.
- **Database Transactions:** Implementing **ACID-compliant transactions** to ensure data consistency and reliability across multiple, interdependent database operations (e.g., borrowing/returning books).
- **Concurrency Handling:** Understanding and implementing basic strategies (like pessimistic locking) to prevent race conditions in multi-user environments.
- **Secure Configuration:** Managing database credentials and connection strings safely using environment variables.
- **Data Validation:** Defining strict data contracts for your API resources using Pydantic.
- **Effective Project Structuring & Dependency Injection:** Building a well-organized and maintainable application.
- **Database Migrations:** Using Alembic to manage database schema evolution gracefully.
- **Unit & Integration Testing:** Building confidence in your code's correctness and preventing regressions through automated tests.

---

3. Core Technologies

- **Python:** 3.10+
- **FastAPI:** Modern, fast (high-performance) web framework for building APIs.
- **Pydantic:** Data validation and settings management using Python type hints.
- **SQLAlchemy:** Python SQL toolkit and Object Relational Mapper (ORM).
- **PostgreSQL:** Robust, open-source relational database.
- **Psycopg2-binary:** PostgreSQL database adapter for Python.
- **Alembic:** Database migration tool for SQLAlchemy.
- **`pytest`**: Python testing framework.
- **`httpx`**: For making HTTP requests in tests (used by FastAPI's `TestClient`).
- **`python-dotenv`**: For loading environment variables.

---

4. Project Requirements & Steps

Step 1: Environment Setup

- **Prerequisite:** A running PostgreSQL server. Using Docker is highly recommended for a quick and isolated setup.
    - Example Docker command: `docker run --name some-postgres -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432 -d postgres`
- Create your project directory (e.g., `libro_api_v2`) and a Python virtual environment inside it.
- Create a `requirements.txt` file with the following, then install them:

```
fastapi
uvicorn[standard]
SQLAlchemy
psycopg2-binary
alembic
python-dotenv
pytest
httpx
# pytest-mock # Optional, if you anticipate complex mocking
```

Install: `pip install -r requirements.txt`

Step 2: Database Configuration (`database.py`)

Goal: Configure the connection to your PostgreSQL database.

Tasks:

- Define your `SQLALCHEMY_DATABASE_URL` using the PostgreSQL connection string format (e.g., `postgresql://user:password@host:port/dbname`).
- **Crucially, manage your database credentials using environment variables** (e.g., `DATABASE_URL` in a `.env` file) – **do not hardcode them.** Use `python-dotenv` to load these.
- Create the SQLAlchemy `engine`, `SessionLocal` class, and declarative `Base`.
    - `Base` is required for your ORM models.
    - `SessionLocal` will provide a database session for each request.
    - Define a `get_db()` dependency for FastAPI to manage sessions.

Step 3: Data Modeling (`models.py`)

Goal: Define the database tables for `Book`, `User`, and `Loan`.

Tasks:

- Import `Base` from `database.py`.
- Create a `Book` class that inherits from `Base`.
    - `id`: Integer, Primary Key
    - `title`: String, not nullable
    - `author`: String, not nullable
    - `isbn`: String (International Standard Book Number), **unique**, not nullable

    - publication_year: Integer, not nullable
    - **total_copies**: Integer, not nullable, default to 1 (new field)
    - **available_copies**: Integer, not nullable, default to 1 (new field, should initially equal total_copies)
    - Add a SQLAlchemy relationship for loans (one-to-many with Loan).
- Create a User class that inherits from Base (simplified).
    - id: Integer, Primary Key
    - name: String, not nullable
    - email: String, **unique**, not nullable
    - Add a SQLAlchemy relationship for loans (one-to-many with Loan).
- Create a Loan class that inherits from Base.
    - id: Integer, Primary Key
    - book_id: Integer, Foreign Key to Book.id, not nullable
    - user_id: Integer, Foreign Key to User.id, not nullable
    - borrow_date: DateTime, not nullable, default to current timestamp
    - due_date: DateTime, not nullable (e.g., 2 weeks after borrow_date)
    - return_date: DateTime, nullable (set when returned)
    - status: String, not nullable (e.g., "borrowed", "returned", "overdue"). Consider using Python's Enum for this.
    - Add SQLAlchemy relationships to book and user (many-to-one).

Step 4: Database Migrations with Alembic

Goal: Initialize Alembic and create your initial database schema.

Tasks:

- Initialize Alembic in your project directory: alembic init alembic
- Modify alembic.ini to point to your database.py's Base for autogenerate.
- Generate your first migration script: alembic revision --autogenerate -m "Initial database setup"
- Review the generated script for Book, User, Loan table creation.
- Apply the migration: alembic upgrade head
- *(Bonus)* If you add a new field later (e.g., genre to Book), repeat the alembic revision --autogenerate and alembic upgrade head steps.

Step 5: API Schemas (schemas.py)

Goal: Define the data shapes for your API requests and responses using Pydantic.

Tasks:

- Create BookBase with title, author, isbn, publication_year, total_copies.
- Create BookCreate inheriting BookBase (used for POST /books). available_copies should be derived from total_copies on creation, not user-provided.
- Create Book inheriting BookBase and adding id, available_copies (used for API responses). Remember to enable orm_mode = True in its Config.
- Create UserBase with name, email.
- Create UserCreate inheriting UserBase.

- Create `User` inheriting `UserBase` and adding `id` (for API responses). `orm_mode = True`.
- Create `LoanBase` with `book_id`, `user_id`, `borrow_date`, `due_date`, `return_date`, `status`.
- Create `LoanCreate` just with `book_id` and `user_id` (for `POST /books/{book_id}/borrow`). The API will set dates and status.
- Create `Loan` inheriting `LoanBase` and adding `id` (for API responses). `orm_mode = True`.
    - *(Consider including nested `Book` and `User` schemas within the `Loan` response for richer data).*
- Define a `BookBorrowRequest` schema that accepts `user_id`. (Sent to `POST /books/{book_id}/borrow`)
- Define a `BookReturnRequest` schema that accepts `loan_id`. (Sent to `POST /loans/{loan_id}/return`)

Step 6: Core Logic & Transactions (`crud.py`)

Goal: Write reusable functions for interacting with the database, especially focusing on transactional operations.

Tasks:

- **Basic CRUD for `Book`:**
    - `create_book(db: Session, book: schemas.BookCreate)`: Initialize `available_copies = book.total_copies`.
    - `get_book(db: Session, book_id: int)`
    - `get_books(db: Session, skip: int = 0, limit: int = 100)`
    - `update_book(db: Session, book_id: int, book_update: schemas.BookCreate)`: Be careful updating `total_copies` to ensure `available_copies` remains consistent.
    - `delete_book(db: Session, book_id: int)`
- **Basic CRUD for `User`:**
    - `create_user(db: Session, user: schemas.UserCreate)`
    - `get_user(db: Session, user_id: int)`
    - `get_users(db: Session, skip: int = 0, limit: int = 100)`
- **Transactional Logic for Loans (`BORROW` & `RETURN` are critical here):**
    - `get_loan(db: Session, loan_id: int)`
    - `get_loans(db: Session, user_id: int = None, book_id: int = None, status: str = None, skip: int = 0, limit: int = 100)`
    - **`borrow_book(db: Session, book_id: int, user_id: int)`:**
        1. **Start Transaction:** Use `with db.begin():` to ensure atomicity.
        2. **Fetch & Lock Book:** Query the `Book` by `book_id` using `.with_for_update()` to prevent race conditions if multiple users try to borrow the same last copy.
        3. **Validate:** Check if the book exists and `book.available_copies > 0`. Raise `HTTPException` (or custom errors) if not.
        4. **Fetch User:** Check if the `User` exists too.
        5. **Decrement `Book.available_copies`**: `book.available_copies -= 1`.

6. **Create Loan Record**: Create a new Loan object, set book_id, user_id, borrow_date, due_date (e.g., current date + 14 days), status='borrowed'.
7. db.add(book) and db.add(new_loan).
8. **Automatic Commit/Rollback:** The with db.begin(): block will automatically commit changes if successful or roll back if any error occurs.

- **return_book(db: Session, loan_id: int):**
  1. **Start Transaction:** with db.begin():.
  2. **Fetch & Lock Loan:** Query the Loan by loan_id using .with_for_update().
  3. **Validate:** Check if the loan exists and loan.status == 'borrowed'. Raise HTTPException if not valid.
  4. **Fetch & Lock Book:** Query the associated Book (via loan.book_id) using .with_for_update(). This ensures the available_copies update is also protected.
  5. **Increment Book.available_copies**: book.available_copies += 1.
  6. **Update Loan Record:** Set loan.status = 'returned' and loan.return_date = datetime.now().
  7. db.add(book) and db.add(loan).
  8. **Automatic Commit/Rollback.**
- *(Consider: get_overdue_loans(db: Session))`

Step 7: API Endpoints (main.py)

Goal: Expose your logic via RESTful API endpoints.

Tasks:

- Set up your FastAPI app and load get_db dependency.
- Implement API endpoints using appropriate HTTP methods and status codes:
  - **Book Endpoints (/books):**
    - POST /books/: Create a new book.
    - GET /books/{book_id}: Read a single book.
    - GET /books/: Read a list of books (with skip/limit pagination).
    - PUT /books/{book_id}: Update a book.
    - DELETE /books/{book_id}: Delete a book.
    - **POST /books/{book_id}/borrow**: Accept schemas.BookBorrowRequest. Call crud.borrow_book. Return schemas.Loan.
  - **User Endpoints (/users):**
    - POST /users/: Create a new user.
    - GET /users/{user_id}: Read a single user.
    - GET /users/: Read a list of users.
  - **Loan Endpoints (/loans):**
    - GET /loans/: Get all loans (with filters for user_id, book_id, status).
    - GET /loans/{loan_id}: Get a specific loan.

        - **POST /loans/{loan_id}/return**: Call crud.return_book. Return schemas.Loan.
- Handle common HTTP errors (e.g., 404 Not Found, 400 Bad Request, 409 Conflict for unique constraints, 412 Precondition Failed for validation errors like "no available copies"). Use FastAPI's HTTPException.

---

8. Unit & Integration Testing

Testing is paramount for ensuring your API behaves as expected, especially when dealing with complex logic like transactions, where data integrity is at stake. You will implement tests for your core business logic (crud.py) and API endpoints (main.py).

Learning Outcomes (Testing Specific):

- **Test-Driven Development (TDD) principles (implicitly):** Though not strictly TDD, you'll learn to think about testable units of code.
- **Pytest:** Get hands-on experience with a powerful and widely used Python testing framework.
- **Fixture Management:** Use pytest fixtures for setting up test environments (e.g., database sessions, test clients).
- **Mocking and Dependency Overriding:** Learn how to isolate components for testing using FastAPI's dependency override mechanism.
- **Testing Database Interactions:** Practice testing functions that interact with a database, ensuring atomicity and consistency of transactions.

Project Requirements & Steps (Testing):

Step 8.1: Create a Test Directory and Setup (tests/conftest.py)

Create a tests/ directory at the root of your project. Inside, create a conftest.py file. This file will define fixtures that can be shared across all your tests.

Goal: Provide isolated, in-memory database sessions and a test client for your API.

Explanation:

- You will use an in-memory SQLite database (sqlite:///:memory:) for unit and integration tests. This is faster than a full PostgreSQL instance and ensures tests are isolated.
- You will use pytest fixtures for database setup and teardown.
- FastAPI's TestClient will be used to make requests to your application within tests.
- You will override FastAPI's get_db dependency to inject your test database session.

Step 8.2: Write Tests for crud.py (Unit Tests)

Create files like tests/test_crud.py. These tests will directly call functions in your crud.py module, passing in the db_session fixture provided by conftest.py.

Goal: Verify that your `crud.py` functions correctly interact with the database (create, read, update, delete) and that transactional logic operates as expected. This includes:

- Successful creation and retrieval of Books, Users, and Loans.
- Correct behavior of `borrow_book` and `return_book` (e.g., `available_copies` updates, `Loan` status changes).
- Error handling and transaction rollback for invalid borrow/return attempts (e.g., trying to borrow an unavailable book, returning an already returned loan).

Step 8.3: Write Tests for `main.py` (Integration Tests)

Create files like `tests/test_api.py`. These tests will use the `client` fixture (from `conftest.py`) to make HTTP requests to your FastAPI application.

Goal: Verify that your API endpoints correctly handle requests, call the underlying `crud` functions, return the correct HTTP status codes, and format responses according to your schemas. This includes:

- Testing all CRUD operations via API calls.
- Testing `POST /books/{book_id}/borrow` and `POST /loans/{loan_id}/return` API endpoints.
- Verifying correct HTTP status codes (200, 201, 400, 404, etc.) for success and failure scenarios.
- Checking the structure and content of the JSON responses.

Step 8.4: Run Your Tests

Navigate to your project root in the terminal and run:

```
Bash

pytest
```

This will discover and run all tests in the `tests/` directory.

Key Testing Principles:

- **Isolation:** Each test should run independently and not affect other tests. This is achieved by using an in-memory database and transactional fixtures that rollback changes after each test.
- **Speed:** In-memory SQLite is fast.
- **Readability:** Tests should be easy to understand using clear naming and the Arrange-Act-Assert pattern.
- **Focus:** Test one specific piece of functionality per test.

---

9. Running and Manual Testing

- Ensure your PostgreSQL server is running.
- Start your FastAPI app: `uvicorn app.main:app --reload`

- Open your browser to `http://127.0.0.1:8000/docs` (or `/redoc`).
- **Use the interactive Swagger UI to test all functionalities manually:**
    - Create a few books and users.
    - Test borrowing a book. Observe `available_copies` changing.
    - Try to borrow the same book when `available_copies` is 0 – expect an error.
    - Test returning a book. Observe `available_copies` changing back and `Loan` status updating.
    - Test getting lists of books, users, and loans.
    - *(Challenge)* Try to simulate a race condition (e.g., rapidly send two borrow requests for the last copy of a book using `curl` or a script in two different terminal windows) to verify your pessimistic locking mechanism works in a real PostgreSQL environment. Does it prevent double-borrowing?

---

10. Deliverables

- **Source Code:** The complete LibroAPI project structured as described, in a Git repository (e.g., GitHub, GitLab).
- **README.md File:** A detailed `README.md` in the repository root, explaining:
    - Project purpose and scope.
    - Setup instructions (PostgreSQL, virtual environment, `requirements.txt`, `.env` example).
    - How to run the application.
    - How to run the tests.
    - Examples for each critical API endpoint (especially borrow/return) on how to use them (e.g., curl commands or Postman/Insomnia instructions).
- **Self-Reflection:** A short summary (can be part of the `README` or a separate document) of what you learned from this project, how the transactional approach addresses the library's data integrity problem, and any challenges you faced and how you overcame them.
