# DAY 23: MORNING ASSESSMENT

# 21. Force the system to handle simultaneous borrow attempts (simulate concurrency with threading)

import threading  
lock = threading.Lock()  
  
def borrow\_book(self, book\_id, user\_id):  
 with lock:  
 if book\_id not in self.books:  
 raise LookupError("Book not found")  
 book = self.books[book\_id]  
 if not book.available:  
 raise BookNotAvailableError("Book already issued")  
 book.available = False  
 self.transactions.append({  
 "book": book\_id,  
 "user": user\_id,  
 "date": datetime.now().isoformat()  
 })  
 self.\_persist\_state()

# 22. Implement input validation: member ID must be alphanumeric, book ID must be unique.

import re  
  
def register\_member(self, user\_id, name):  
 if not re.match(r"^[A-Za-z0-9]+$", user\_id):  
 raise ValueError("Member ID must be alphanumeric")  
 if user\_id in self.users:  
 raise ValueError("User already exists")  
 self.users[user\_id] = User(user\_id, name)  
 self.\_persist\_state()  
  
def add\_library\_book(self, book\_id, title, author, isbn):  
 if book\_id in self.books:  
 raise ValueError("Book ID must be unique")  
 self.books[book\_id] = Book(book\_id, title, author, isbn)  
 self.\_persist\_state()

# 23. Raise an exception if a member tries to borrow more than 5 books at once.

def borrow\_book(self, book\_id, user\_id):  
 borrowed = [t for t in self.transactions if t["user"] == user\_id and "return\_date" not in t]  
 if len(borrowed) >= 5:  
 raise Exception("Member cannot borrow more than 5 books")  
 # proceed to issue  
 if book\_id not in self.books:  
 raise LookupError("Book not found")  
 book = self.books[book\_id]  
 if not book.available:  
 raise BookNotAvailableError("Book already issued")  
 book.available = False  
 self.transactions.append({  
 "book": book\_id,  
 "user": user\_id,  
 "date": datetime.now().isoformat()  
 })  
 self.\_persist\_state()

# 24. Implement retry logic if the file is locked when saving.

import time  
  
def \_persist\_state(self):  
 retries = 3  
 for i in range(retries):  
 try:  
 with open(STORE, "w", encoding="utf-8") as f:  
 data = {  
 "books": [b.\_\_dict\_\_ for b in self.books.values()],  
 "users": [u.\_\_dict\_\_ for u in self.users.values()],  
 "transactions": self.transactions  
 }  
 json.dump(data, f, indent=4)  
 break  
 except PermissionError:  
 logging.warning("File locked, retrying...")  
 time.sleep(1)  
 else:  
 logging.error("Failed to save after retries")

# 25. Add versioning to the JSON file, so each save creates a backup copy.

import shutil  
  
def \_persist\_state(self):  
 if os.path.exists(STORE):  
 backup = f"{STORE}\_{datetime.now().strftime('%Y%m%d%H%M%S')}.bak"  
 shutil.copy(STORE, backup)  
 with open(STORE, "w", encoding="utf-8") as f:  
 data = {  
 "books": [b.\_\_dict\_\_ for b in self.books.values()],  
 "users": [u.\_\_dict\_\_ for u in self.users.values()],  
 "transactions": self.transactions  
 }  
 json.dump(data, f, indent=4)

# 26. Use append-only log file for all actions.

def log\_action\_lib(self, action):  
 with open("actions.log", "a", encoding="utf-8") as f:  
 f.write(f"{datetime.now().isoformat()} - {action}\n")  
  
# Example usage after a successful borrow:  
# self.log\_action\_lib(f"Book issued: {book\_id} by {user\_id}")

# 27. Add an import/export feature (JSON ↔ TXT ↔ CSV).

import csv  
  
def export\_books\_csv(self, filename="library.csv"):  
 with open(filename, "w", newline="", encoding="utf-8") as f:  
 writer = csv.writer(f)  
 writer.writerow(["BookID", "Title", "Author", "ISBN", "Available"])  
 for b in self.books.values():  
 writer.writerow([b.book\_id, b.title, b.author, b.isbn, b.available])  
  
def export\_books\_txt(self, filename="library.txt"):  
 with open(filename, "w", encoding="utf-8") as f:  
 for b in self.books.values():  
 f.write(f"{b.book\_id} - {b.title} - {b.author}\n")  
  
def import\_books\_json(self, filename="library.json"):  
 with open(filename, "r", encoding="utf-8") as f:  
 data = json.load(f)  
 self.books = {b["book\_id"]: Book(\*\*b) for b in data["books"]}

# 28. Store last modified timestamp of each book inside the JSON data.

class Book:  
 def \_\_init\_\_(self, book\_id, title, author, isbn, available=True):  
 self.book\_id = book\_id  
 self.title = title  
 self.author = author  
 self.isbn = isbn  
 self.available = available  
 self.last\_modified = datetime.now().isoformat()  
  
 def update(self, title=None, author=None):  
 if title:  
 self.title = title  
 if author:  
 self.author = author  
 self.last\_modified = datetime.now().isoformat()

# 29. Use pickle for faster serialization of the entire library state.

import pickle  
  
def save\_state\_pickle(self, filename="library.pkl"):  
 with open(filename, "wb") as f:  
 pickle.dump(self, f)  
  
@staticmethod  
def load\_state\_pickle(filename="library.pkl"):  
 with open(filename, "rb") as f:  
 return pickle.load(f)

# 30. Implement a sliding-scale fine calculator.

def compute\_fine(self, borrow\_date, return\_date):  
 days = (return\_date - borrow\_date).days  
 if days <= 14:  
 return 0  
 late = days - 14  
 if late <= 5:  
 return late \* 2  
 elif late <= 10:  
 return (5 \* 2) + (late - 5) \* 5  
 else:  
 return (5 \* 2) + (5 \* 5) + (late - 10) \* 10