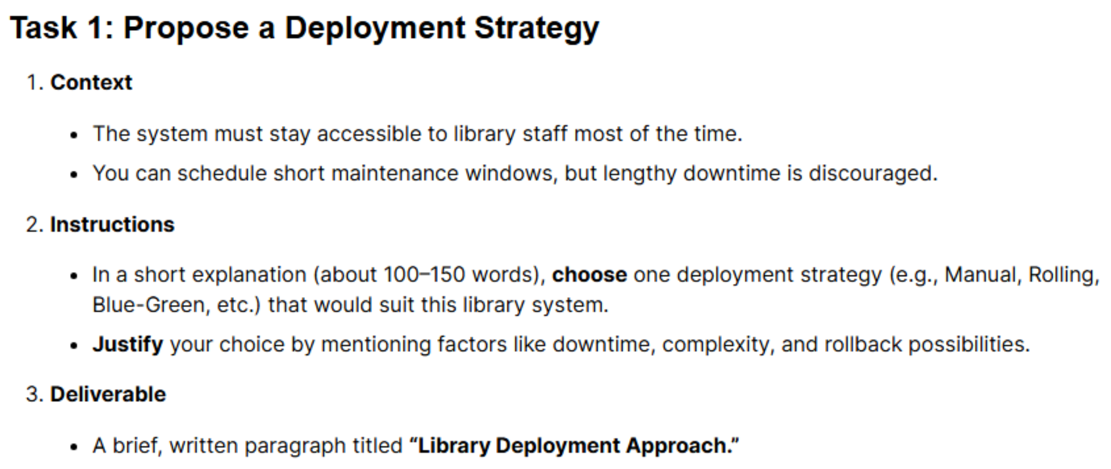
TASKS

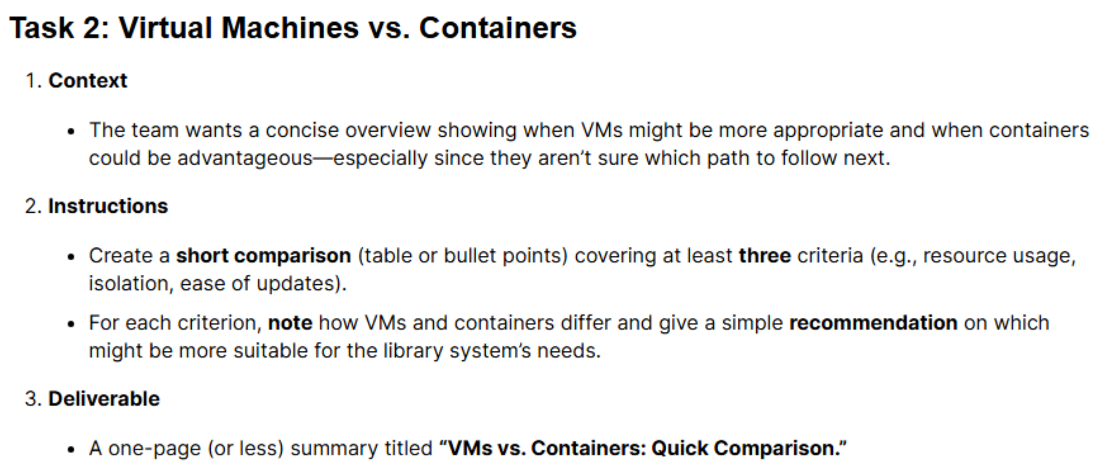


Task 1: Propose a Deployment Strategy

1. **Context**
   - The system must stay accessible to library staff most of the time.
   - You can schedule short maintenance windows, but lengthy downtime is discouraged.
2. **Instructions**
   - In a short explanation (about 100–150 words), **choose** one deployment strategy (e.g., Manual, Rolling, Blue-Green, etc.) that would suit this library system.
   - **Justify** your choice by mentioning factors like downtime, complexity, and rollback possibilities.
3. **Deliverable**
   - A brief, written paragraph titled **"Library Deployment Approach."**

Library Deployment Approach

I propose using the **Rolling Deployment strategy**. This approach involves gradually replacing the older version of the application with the new version on a set of servers (or containers) one by one. This minimizes downtime, as the old version remains operational until the new one is confirmed to be running correctly. The library system requires high availability with minimal tolerance for downtime, making the zero-downtime nature of rolling deployment ideal. While slightly more complex than a manual update, it is less complex than Blue-Green. A key advantage is the possibility of an easy rollback: if the new version encounters issues, the process can be halted, and traffic can be immediately rerouted back to the untouched, running older instances, ensuring library staff can continue their work quickly.



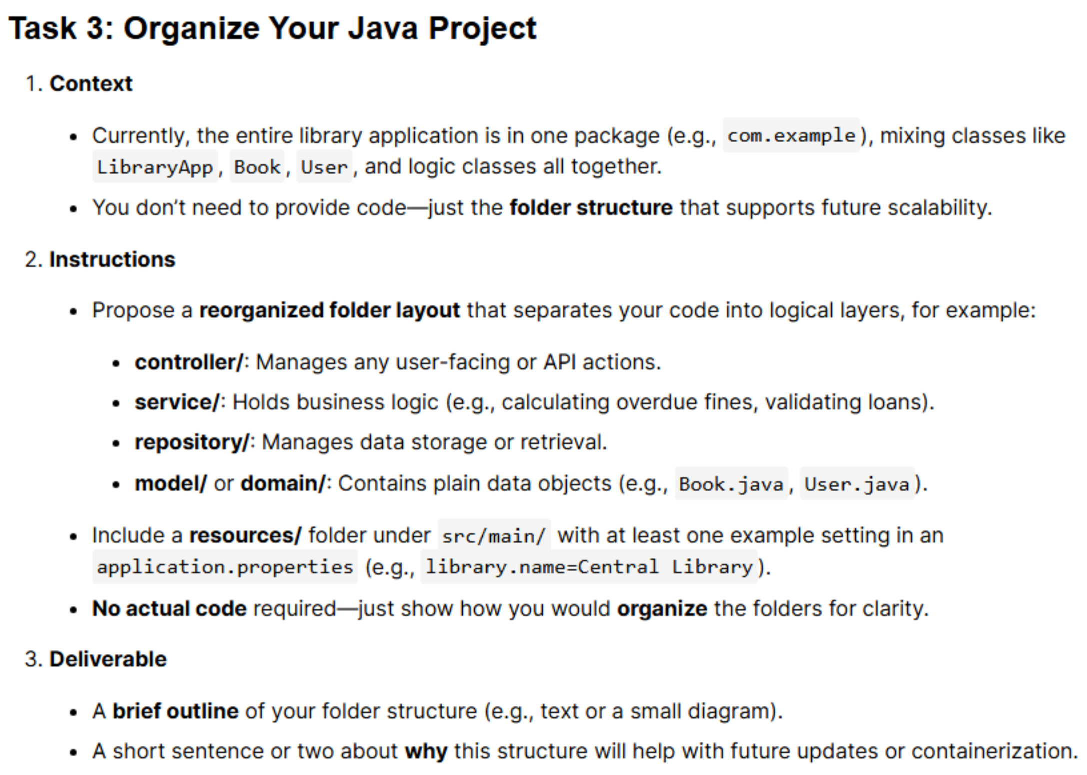
Task 2: Virtual Machines vs. Containers

1. **Context**
   - The team wants a concise overview showing when VMs might be more appropriate and when containers could be advantageous—especially since they aren't sure which path to follow next.
2. **Instructions**
   - Create a **short comparison** (table or bullet points) covering at least **three** criteria (e.g., resource usage, isolation, ease of updates).
   - For each criterion, **note** how VMs and containers differ and give a simple **recommendation** on which might be more suitable for the library system's needs.
3. **Deliverable**
   - A one-page (or less) summary titled **"VMs vs. Containers: Quick Comparison."**

VMs vs. Containers: Quick Comparison

Criterion	Virtual Machines (VMs)	Containers (e.g., Docker)	Suitability for Library System
Resource Usage	**High**. Each VM requires its own full OS, leading to significant overhead in CPU, RAM, and disk space.	**Low**. Share the host OS kernel, only packaging the application and its dependencies. Extremely lightweight and fast to start.	**Containers**. Lower resource consumption means lower hosting costs and better performance for a small application.
Isolation	**Strong (OS-level)**. Completely isolated from the host OS and other VMs, providing maximum security and independence.	**Moderate (Process-level)**. Isolated from other containers, but they all share the host OS kernel. Isolation is less robust than VMs.	**Containers**. Strong isolation isn't strictly necessary for a non-security-critical internal application, and the resource benefits outweigh the slight isolation difference.
Ease of Updates	**Low/Slow**. Requires updating the application, libraries, and potentially the entire Guest OS on each VM. Image creation is slow.	**High/Fast**. Updates only the application code/dependencies within the container image. New containers can be built and deployed in seconds.	**Containers**. Given frequent updates (every few weeks), the speed and consistency of container updates are a major operational advantage.

Recommendation: **Containerization** is the superior choice for the Library Management System. Its low resource usage, rapid deployment cycle, and ease of updates align perfectly with a small, frequently-updated application requiring high availability.



Task 3: Organize Your Java Project

1. **Context**
   - Currently, the entire library application is in one package (e.g., `com.example`), mixing classes like `LibraryApp`, `Book`, `User`, and logic classes all together.
   - You don't need to provide code—just the **folder structure** that supports future scalability.
2. **Instructions**
   - Propose a **reorganized folder layout** that separates your code into logical layers, for example:
     - **controller/**: Manages any user-facing or API actions.
     - **service/**: Holds business logic (e.g., calculating overdue fines, validating loans).
     - **repository/**: Manages data storage or retrieval.
     - **model/** or **domain/**: Contains plain data objects (e.g., `Book.java`, `User.java`).
   - Include a **resources/** folder under `src/main/` with at least one example setting in an `application.properties` (e.g., `library.name=Central Library`).
   - **No actual code** required—just show how you would **organize** the folders for clarity.
3. **Deliverable**
   - A **brief outline** of your folder structure (e.g., text or a small diagram).
   - A short sentence or two about **why** this structure will help with future updates or containerization.

📁 Java Folder Structure Outline

This structure follows a typical layered (or MVC/Microservice) architecture, promoting the Separation of Concerns

```
LibraryManagementSystem/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── library/
│   │   │           ├── LibraryApp.java        <-- Main application entry point
│   │   │           ├── controller/            <-- REST APIs, User Interface logic
│   │   │           ├── service/               <-- Business Logic (e.g., LoanServic
│   │   │           ├── repository/            <-- Data access layer (e.g., UserRep
│   │   │           └── model/                 <-- Plain data objects (POJOs) (e.g.
│   │   └── resources/
│   │       ├── application.properties
│   │       └── log4j.properties
│   └── test/
│       └── java/
│           └── com/
│               └── library/
│                   ├── service/
│                   └── repository/
│
├── target/                                    <-- Compiled JAR/WAR file output
├── pom.xml                                    <-- Maven/Gradle build file
└── README.md
```

Example application.properties content:

Properties

```
library.name=Central Library
database.url=jdbc:postgresql://localhost:5432/librarydb
server.port=8080
```

Justification:

This layered structure ensures that business logic (service) is fully isolated from the data access (repository) and user-facing (controller) concerns. This separation makes the system significantly easier to maintain, as changes in one layer (e.g., switching databases only impacts the repository) won't break others. Furthermore, this clean organization is crucial for containerization, as it simplifies dependencies and clearly delineates the application components needed inside the final container image, promoting clean and efficient JAR/WAR file creation.