# **High-Level Design for "Amazing Books" Portal**

To achieve scalability, resiliency, and fast-paced incremental releases, a **Microservices Architecture** is proposed. Below is the high-level design that breaks down the system into services, databases, communication flows, and considerations for non-functional requirements.

# Presented By: Praveen K. Srivastava

# Table of Contents

**1. Key Microservices**  
 1.1 Book Management Service (bookms)  
 1.2 Issue Management Service (issuerms)  
 1.3 Authentication Service (authms)  
 1.4 Service Discovery Service  
  
2. System Architecture  
 2.1 Microservices Components  
  
3. API Design  
 3.1 Book Management Service (bookms)  
 3.2 Issue Management Service (issuerms)  
 3.3 Authentication Service (authms)  
  
4. Technology Stack  
  
5. Scalability  
  
6. Resiliency  
  
7. Security  
  
8. Performance  
  
9. Quick Releases  
  
10. Deployment Model

# **Key Microservices**

## **Book Management Service (bookms):**

* + Responsible for managing books (CRUD operations).
  + Exposes APIs to add, edit, delete, and fetch books.
  + Maintains a database (bookdb) with the following schema:
    - isbn: String (Primary Key)
    - title: String
    - publishedDate: Date
    - totalCopies: Integer
    - issuedCopies: Integer
    - author: String

## **Issue Management Service (issuerms):**

* + Handles book issuance to customers.
  + Fetches book details from bookms using RESTful APIs.
  + Updates the issuedCopies count in bookms.
  + Manages customer-book relationships.
  + Maintains a database (issuedb) with the following schema:
    - isbn: String (Foreign Key)
    - custId: String
    - noOfCopies: Integer

## **Authentication Service (authms):**

* + Implements OAuth2 for user authentication and authorization.
  + In-memory storage for managing customers and their roles.
  + Generates access tokens for authenticated API calls.

## **Service Discovery Service:**

* + Manages dynamic service registration and discovery.
  + Enables automatic detection of available services for scaling.
  + **Technology:** Use a service registry like **Eureka** (Spring Cloud) or **Consul**.

# **System Architecture**

## **Microservices Components**

1. **REST APIs**:  
   Each service exposes APIs that adhere to RESTful principles for seamless integration with GUIs and devices.
2. **Database**:
   * **H2 Database** (In-memory for development; can migrate to persistent DB in production, e.g., PostgreSQL, MySQL).
   * Each service has its own database to follow **Database-per-Microservice** design.
3. **Service Communication**:
   * Use **REST APIs** for synchronous communication between services.
   * For asynchronous communication, use a **message broker** like **RabbitMQ** or **Kafka** (for resiliency and decoupling).
4. **Service Discovery**:  
   A service registry helps dynamically manage service instances, supporting horizontal scaling.

# **API Design**

**1. Book Management Service (bookms):**

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /books | Fetch all books |
| GET | /books/{isbn} | Fetch book by ISBN |
| POST | /books | Add a new book |
| PUT | /books/{isbn} | Update book details |
| DELETE | /books/{isbn} | Delete a book |

**2. Issue Management Service (issuerms):**

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /issues/{custId} | Fetch books issued to a customer |
| POST | /issues | Issue a book to a customer |
| DELETE | /issues/{custId}/{isbn} | Cancel an issued book for a customer |

**3. Authentication Service (authms):**

| **Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| POST | /oauth/token | Generate OAuth2 token |
| GET | /users | Fetch all registered users |
| POST | /users | Register a new user |

# **Technology Stack**

* **Framework**: Spring Boot
* **Database**: H2 (for development; migrate to PostgreSQL for production)
* **Authentication**: OAuth2 with Spring Security
* **Service Discovery**: Eureka or Consul
* **Message Broker**: RabbitMQ or Kafka (for resiliency)
* **API Gateway**: Spring Cloud Gateway for routing and load balancing

# **Scalability**

* **Horizontal Scaling**:
  + Each service can scale independently based on load.
  + Use a **load balancer** (e.g., NGINX) or Kubernetes for managing scalability.
* **Database Optimization**:
  + For production, use a persistent database with partitioning or replication.

# **Resiliency**

1. **Retry Mechanisms**: Use retry patterns for failed inter-service communication.
2. **Circuit Breakers**: Use libraries like **Resilience4j** to handle service failures gracefully.
3. **Failover Mechanisms**: Ensure that services are deployed in a cluster for high availability.

# **Security**

1. **OAuth2 Authentication**: All APIs are secured with access tokens.
2. **Role-Based Access Control**: Ensure certain APIs are restricted to authorized roles.
3. **Data Encryption**: Encrypt sensitive data at rest and in transit.

# **Performance**

1. **Caching**: Use a caching layer (e.g., Redis) to store frequently accessed data like book details.
2. **Asynchronous Processing**: Use message queues for non-critical, time-consuming operations like notifications.

# **Quick Releases**

* **CI/CD Pipeline**: Automate builds, tests, and deployments with tools like Jenkins, GitHub Actions, or GitLab CI.
* **Containerization**: Package each microservice in Docker containers for consistent deployments.

# **Deployment Model**

* **Local Development**: Use H2 databases and run services on local ports.
* **Staging/Production**:
  + Deploy using Docker and orchestrate with Kubernetes.
  + Use persistent databases (e.g., PostgreSQL).

This architecture ensures the portal is scalable, secure, and easily extendable for future features.