1. Explain which SOA design principles you applied when decomposing the monolith into independent services.

**## 1. Service Autonomy 🔑**

**Service Autonomy** means that a service has full control over the logic it encapsulates, the data it manages, and the technology it's built on.

* **How it was applied:** Each of the four services—**Catalog, Orders, Payments, and Shipping**—was designed with its **own private data store**. The Catalog service manages product information in its database, and no other service can access that database directly. If the Orders service needs product details, it must ask the Catalog service through its public interface.
* **Why it helps:** This is the most critical principle for solving GlobalBooks' original problems. Because the Payments service is autonomous, its development team can add a new payment provider, test it, and deploy it **without touching or redeploying the Catalog or Shipping services**. This drastically reduces deployment risks and downtime. It also allows us to scale services independently; if holiday promotions cause a surge in traffic to the Catalog service, we can allocate more resources specifically to it without over-provisioning the Payments service.

**## 2. Standardized Service Contract 📜**

This principle dictates that services must adhere to a formal, published contract that describes their purpose, operations, and data requirements. The contract is the only entry point to the service's logic.

* **How it was applied:** Each service exposes its functionality through well-defined interfaces. We used **SOAP (with WSDL)** for legacy partners who require it and **REST (with an OpenAPI specification)** for modern web and mobile clients. For example, the Payments service contract clearly defines an operation like processPayment that requires a specific set of inputs (e.g., orderID, amount, creditCardInfo) and guarantees a specific output (e.g., transactionID, status). The internal Java code that actually performs the payment processing is hidden from all consumers.
* **Why it helps:** This abstraction separates the "what" (the service's capability) from the "how" (the implementation). It allows the underlying logic of a service to change without breaking the clients that depend on it, as long as the contract remains stable.

**## 3. Service Loose Coupling 🔗**

Loose coupling aims to minimize the dependencies between services. A change in one service should have little to no impact on the other services that communicate with it.

* **How it was applied:** The primary tool for achieving loose coupling in this design is the **RabbitMQ ESB (Enterprise Service Bus)**. When a customer places an order, the Orders service doesn't directly and synchronously call the Payments and Shipping services. Instead, it publishes an OrderPlaced event to the RabbitMQ message broker.
* **Why it helps:** This asynchronous, event-driven communication is a game-changer for resilience. If the Shipping service is temporarily down for maintenance, new orders can still be successfully placed and paid for. The OrderPlaced message simply waits in a queue until the Shipping service comes back online to process it. This prevents a failure in one minor service from causing a cascade failure across the entire system, a key weakness of the original monolith.

**## 4. Service Composability 🧩**

This principle ensures that existing services can be coordinated and assembled to create more complex, higher-level business processes.

* **How it was applied:** The "PlaceOrder" workflow is not a single service but a **composition** of the other, more granular services. We used a **BPEL (Business Process Execution Language) engine** to orchestrate this workflow. The BPEL process defines the logic:
  1. Receive the initial order request.
  2. Call the createOrder operation on the **Orders service**.
  3. Upon success, simultaneously call the processPayment operation on the **Payments service** and the verifyInventory operation on the **Catalog service**.
  4. If both succeed, call the scheduleShipment operation on the **Shipping service**.
* **Why it helps:** This approach makes the business logic explicit and adaptable. If GlobalBooks wants to introduce a new business process, like "Express Order with Gift Wrapping," it can be created by composing the existing services in a new BPEL workflow without writing significant new code. This directly supports business agility.

**2** Discuss one key benefit and one primary challenge of your approach.

#### Key Benefit: Improved Scalability

A primary benefit of transforming the monolithic architecture into a Service-Oriented Architecture (SOA) is significantly improved scalability. In an e-commerce platform like GlobalBooks, certain parts of the system experience dramatically higher loads during specific periods, such as holiday seasons or promotional flash sales.

Under the SOA model, services experiencing high demand, such as the OrdersService and PaymentsService, can be scaled independently. This means resources can be allocated precisely where they are needed without scaling the entire application. This approach ensures more efficient use of infrastructure resources and reduces operational costs, directly addressing the risk of system-wide failures that the original monolith faced during peak traffic.

#### Key Challenge: Operational Complexity

However, a key challenge introduced by this distributed approach is the increase in operational complexity. Managing a single monolithic application is far simpler than managing a distributed system composed of multiple independent services.

With the new architecture, we must deploy, monitor, log, and debug four separate services (Catalog, Orders, Payments, Shipping), their respective data stores, and the communication network between them. Tasks like troubleshooting an issue that spans across multiple services or ensuring data consistency for a single transaction that involves several services (e.g., placing an order) become significantly more complex. This requires robust monitoring tools and a well-defined strategy for distributed transaction management.

3 Provide a WSDL excerpt for the CatalogService (operations, types, binding)

### Explanation of WSDL Components

1. **<types>**: This section defines the data structures. We define the requests and responses for our operations here. For example, GetBookDetailsResponse is defined to return a book's Title, Author, Price, and Stock Quantity.
2. **<message>**: These are the "envelopes" that carry the data defined in <types>. Each operation needs an input message (for the request) and an output message (for the response).
3. **<portType>**: This is an abstract definition of the service's operations. It's like a Java interface; it lists the functions (GetBookDetails, UpdateStock) and the messages they use, but it doesn't say *how* they are implemented.
4. **<binding>**: This section gets concrete. It specifies *how* the operations in the <portType> will be transmitted. Here, we bind them to the **SOAP protocol** over **HTTP**. We use style="document" and use="literal", which is a common and straightforward standard.
5. **<service>**: This is the final piece. It provides the actual physical address (URL endpoint) where the service can be accessed. Client applications will send their SOAP requests to this URL.

**4** Draft the UDDI registry entry metadata enabling client discovery.

#### 1. Contract-First Development

Contract-first is a development methodology where the service contract is defined *before* any implementation code is written. In the context of our CatalogService, the **WSDL file serves as this formal contract**.

**How it works:**

1. **Define the Contract:** First, as we have just done, we create the CatalogService.wsdl file. This document clearly defines the service's operations (GetBookDetails, UpdateStock), the exact message formats, and the data types involved. All teams agree on this contract before development begins.
2. **Enable Parallel Development:** With the WSDL contract finalized, development can proceed in parallel, significantly speeding up the project timeline:
   * **The Backend Team** can use the WSDL to automatically generate server-side skeleton code (e.g., Java interfaces and classes). They can then focus on implementing the core business logic for fetching book details from the database and updating stock levels.
   * **The Client Team** (which could be the team building the OrdersService or a web frontend) can use the same WSDL to generate client-side stubs. This allows them to start building and testing their application's logic for calling the CatalogService without having to wait for the backend implementation to be complete. They can use mock data that conforms to the WSDL contract.
3. **Guarantee Compatibility:** Since both the service provider (backend) and the service consumer (client) build their code based on the exact same contract, it ensures that the two systems will integrate smoothly once development is complete, minimizing integration issues.

### 2. Service Discovery

Service discovery is the process by which a client application can dynamically find and locate a service on a network and understand how to interact with it. This is crucial in large enterprise systems where services may be numerous and their locations might change.

**How WSDL enables this with UDDI:**

1. **Publishing:** The CatalogService.wsdl file is published to a central registry known as a **UDDI (Universal Description, Discovery, and Integration) registry**. A UDDI registry acts like a "Yellow Pages" for web services, categorizing them and storing information about them.
2. **Discovery:** When another application, such as the OrdersService, needs to find information about a book, it can query the UDDI registry. It might search for services categorized under "catalog" or "inventory management".
3. **Retrieving the Contract:** Upon finding the CatalogService in the registry, the client application retrieves its associated CatalogService.wsdl file.
4. **Invoking the Service:** By parsing the WSDL file, the client application can automatically learn everything it needs to know to interact with the service: its physical location (the endpoint URL from the <service> tag), the available operations (<portType>), and the precise XML structure of the request and response messages (<types> and <message>). This allows for dynamic binding, where a client can find and use a service at runtime without being hard-coded to its location.

5 Describe in detail how you implemented the CatalogService SOAP endpoint in Java (including sun jaxws.xml and web.xml snippets).

7 Design the OrdersService REST API: list endpoints (POST /orders, GET /orders/{id}), sample JSON request & response, and the JSON Schema for order creation.

### Task 7: OrdersService REST API Design

#### 1. Introduction

The OrdersService is a crucial component of the GlobalBooks microservices architecture. It exposes a RESTful API to handle all operations related to customer orders. This document outlines the API design, including its endpoints, sample data structures, and the validation schema for creating new orders. The API uses JSON (JavaScript Object Notation) for data interchange.

#### 2. API Endpoint Design

The following endpoints have been defined for the OrdersService:

| HTTP Method | Endpoint URL | Description |
| --- | --- | --- |
| POST | /orders | Creates a new customer order. The request body must contain the customer ID, order items, and shipping address. |
| GET | /orders/{id} | Retrieves the details of a specific order using its unique orderId. |

Export to Sheets

#### 3. Sample Interaction

This section demonstrates a typical request-response cycle for creating a new order.

##### 3.1. Sample Request Body for POST /orders

To create a new order, the client sends a POST request to the /orders endpoint with the following JSON payload:

JSON

{

"customerId": "CUST-789",

"orderItems": [

{

"isbn": "978-0321765723",

"quantity": 1,

"unitPrice": 25.99

},

{

"isbn": "978-0743273565",

"quantity": 2,

"unitPrice": 15.50

}

],

"shippingAddress": {

"street": "123 Galle Road",

"city": "Colombo",

"postalCode": "00300",

"country": "Sri Lanka"

}

}

##### 3.2. Sample Response Body

Upon successful creation of the order, the API responds with a 201 Created status code and a JSON body containing the newly assigned orderId, the order status, and a summary of the order details.

JSON

{

"orderId": "ORD-1001",

"status": "PENDING",

"customerId": "CUST-789",

"totalAmount": 57.00,

"orderDate": "2025-09-02T10:30:00Z",

"orderItems": [

{

"isbn": "978-0321765723",

"quantity": 1

},

{

"isbn": "978-0743273565",

"quantity": 2

}

]

}

#### 4. Data Validation with JSON Schema

To ensure the integrity of incoming data, a JSON Schema is used to validate the payload of every new order creation request (POST /orders). This schema defines the structure, data types, and constraints for the request body.

##### 4.1. JSON Schema Definition

JSON

{

"$schema": "http://json-schema.org/draft-07/schema#",

"title": "New Order Creation",

"description": "Schema for validating a new order request",

"type": "object",

"properties": {

"customerId": {

"description": "Unique identifier for the customer",

"type": "string",

"minLength": 1

},

"orderItems": {

"description": "List of items included in the order",

"type": "array",

"minItems": 1,

"items": {

"type": "object",

"properties": {

"isbn": { "type": "string", "description": "The 13-digit ISBN of the book" },

"quantity": { "type": "integer", "description": "Number of units for this item", "minimum": 1 },

"unitPrice": { "type": "number", "description": "The price of a single unit of the item", "exclusiveMinimum": 0 }

},

"required": ["isbn", "quantity", "unitPrice"]

}

},

"shippingAddress": {

"description": "The address where the order should be shipped",

"type": "object",

"properties": {

"street": { "type": "string" },

"city": { "type": "string" },

"postalCode": { "type": "string" },

"country": { "type": "string" }

},

"required": ["street", "city", "country"]

}

},

"required": [

"customerId",

"orderItems",

"shippingAddress"

]

}

**Key validation rules enforced by this schema include:**

* The customerId, orderItems, and shippingAddress fields are mandatory.
* The orderItems array must contain at least one item.
* Each item in the orderItems array must have a valid isbn, a quantity of at least 1, and a unitPrice greater than 0.

8 Outline the “PlaceOrder” BPEL process: receive, loop for price lookup via CatalogService, invoke OrdersService, reply to client.