**Task: Distributed Inventory and Order Management System**

**Scenario:** You are tasked with building an **Inventory and Order Management System**. This system allows users to create orders, check product availability, and reserve inventory. Multiple services must communicate **synchronously** with each other.

### Microservices:

1. **Product Service**: Manages product details and availability.
2. **Inventory Service**: Manages inventory levels and reservations.
3. **Order Service**: Manages orders, communicates with both Product and Inventory services.
4. **Payment Service**: Processes payments and communicates with the Order Service.

**Microservice 1: Product Service**

1. **Endpoints**:
   * GET /products: Fetch all available products.
   * GET /products/{id}: Fetch details of a specific product.
   * POST /products: Add a new product.
2. **Business Logic**:
   * Products have fields such as id, name, price, and available (boolean to indicate availability).
   * A product can only be marked as "available" if the **Inventory Service** has enough stock for it.
   * Validate all product information on creation (e.g., price cannot be negative).

**Microservice 2: Inventory Service**

1. **Endpoints**:
   * GET /inventory/{productId}: Get available stock for a product.
   * POST /inventory/reserve: Reserve stock for a product.
   * POST /inventory/cancel-reservation: Cancel a reservation if something goes wrong in the order flow.
2. **Business Logic**:
   * Inventory should track the stock for each product (e.g., productId, stock).
   * For each order request, reserve stock if it's available and notify the **Order Service**.
   * If stock is insufficient, return an error response to the **Order Service**.
   * Once the order is confirmed, reduce the available stock permanently. If the payment fails, release the reservation by calling cancel-reservation.

**Microservice 3: Order Service**

1. **Endpoints**:
   * POST /orders: Create an order.
   * GET /orders/{orderId}: Fetch details of an order.
   * PUT /orders/{orderId}/cancel: Cancel an order (and release inventory).
2. **Business Logic**:
   * Users can place an order by providing a product ID and quantity.
   * Upon receiving an order request, the **Order Service** should:
     + First, check with the **Product Service** if the product exists and is available.
     + If the product is available, call the **Inventory Service** to reserve stock.
     + Once the stock is reserved, proceed to the payment step by calling the **Payment Service**.
     + If the payment fails or there is any issue during the process, the reserved inventory should be released (rollback mechanism).
     + If everything goes smoothly, mark the order as CONFIRMED. If there’s a failure, mark it as CANCELED.
   * Orders have states: PENDING, CONFIRMED, CANCELED.
   * Implement **cascading rollback**: If a failure occurs at any point in the chain (inventory check, stock reservation, payment), undo all changes made (e.g., cancel inventory reservation and mark the order as failed).

**Microservice 4: Payment Service**

1. **Endpoints**:
   * POST /payments: Process payment for an order.
   * GET /payments/{orderId}: Get the payment status of a specific order.
2. **Business Logic**:
   * Simulate payment processing by generating random outcomes (e.g., 90% chance of success, 10% chance of failure).
   * If the payment is successful, notify the **Order Service** to confirm the order.
   * If the payment fails, notify the **Order Service** to cancel the order and roll back the inventory reservation.
   * Payment statuses: SUCCESS, FAILED, PENDING.

**Synchronous Communication:**

1. **RestTemplate** and **FeignClient** must be used for communication between the services.
   * **RestTemplate**: Use it for communication between **Order Service** and **Product Service**.
   * **FeignClient**: Use it for communication between **Order Service**, **Inventory Service**, and **Payment Service**.
2. All communication between microservices should be **synchronous**. If a service is down or fails, proper error handling should occur in each service, with a meaningful HTTP response code (e.g., 4xx for client errors, 5xx for server errors).
3. Implement **retry logic**: If a microservice call fails, retry it 3 times before marking the order or operation as failed. Use Spring's @Retryable for this.

### ****Fault Tolerance and Rollback (Transaction Management)****

1. Implement a **manual rollback mechanism** in the **Order Service**. If a failure occurs (e.g., payment fails, inventory reservation fails), undo all previous operations.
   * If stock is reserved, but payment fails, call the **Inventory Service** to cancel the reservation.
   * If a failure happens during product availability check or stock reservation, immediately return an error and stop the flow.
2. **Consistency and Error Handling**:
   * Use proper status codes for all failures.
   * In case of failure, all modifications (inventory reservation, order creation) should be rolled back to avoid inconsistent data.

### ****Edge Cases****

1. **Concurrency Handling**: Ensure that simultaneous order requests for the same product don't cause race conditions in the **Inventory Service**. Lock the product during stock reservation to prevent race conditions.
2. **Retry and Timeout Handling**:
   * Implement retry logic for network issues (e.g., if the **Payment Service** is down, retry the payment up to 3 times).
   * Set timeouts for all **RestTemplate** and **FeignClient** calls to avoid hanging requests.
3. **Order Cancellations**: Allow users to cancel orders, which should automatically trigger the release of reserved stock.

### Example Flow:

1. A user calls POST /orders on the **Order Service** with a product ID and quantity.
2. The **Order Service** synchronously checks with the **Product Service** if the product is available using RestTemplate.
3. If the product is available, the **Order Service** sends a request to the **Inventory Service** using FeignClient to reserve stock.
4. Once the stock is reserved, the **Order Service** calls the **Payment Service** using FeignClient to process the payment.
5. If the payment is successful, the **Order Service** confirms the order and marks it as CONFIRMED.
6. If any service (product check, stock reservation, or payment) fails, the **Order Service** triggers a rollback by canceling the inventory reservation and marking the order as CANCELED.