**Exercise 10: Online Bookstore - Configuring Content Negotiation**

Business Scenario:

Support different media types (JSON, XML) for your bookstore's RESTful services.

**. Introduction**

In this exercise, you implemented content negotiation in a Spring Boot application to support different media types, such as JSON and XML. The goal was to configure the application to handle requests and responses in different formats based on the client's preference, specified in the Accept header.

**2. Content Negotiation Configuration**

Content negotiation is a mechanism that allows clients to request data in different formats. To enable this in Spring Boot, you configured content negotiation strategies.

Key points of the configuration:

* **@EnableWebMvc**: Enables Spring MVC configuration and allows for customization of default settings.
* **ContentNegotiationConfigurer**: Used to specify the content negotiation strategy. In this case, you enabled content negotiation using the favorPathExtension(false) method, which disables path extension-based negotiation, and favorParameter(true), which allows for media type specification through a request parameter (e.g., format=json or format=xml).
* **Default Media Type**: You set a default media type, MediaType.APPLICATION\_JSON, so that if no media type is specified by the client, JSON is used.

**Producing and Consuming Different Media Types**

To handle content negotiation in your controllers, you annotated methods with @RequestMapping or @GetMapping and specified the produces and consumes attributes to indicate which media types the method can handle.

Key points:

* **produces**: Specifies the media types that the method can produce in response. For example, you configured the method to produce both JSON and XML responses based on the Accept header.
* **consumes**: Specifies the media types that the method can consume, allowing it to handle requests in both JSON and XML formats.

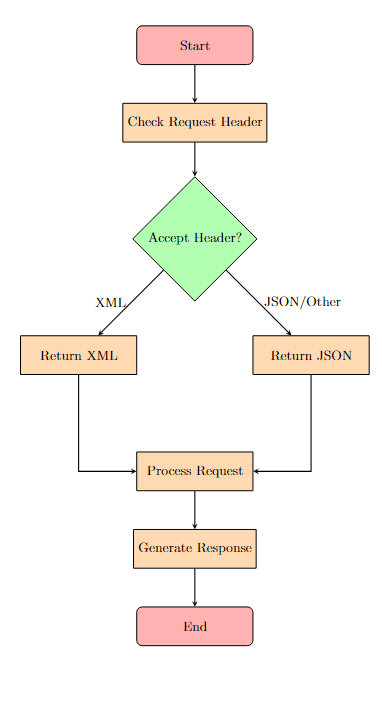
**4. Using Accept Header**

In content negotiation, the Accept header of the HTTP request determines the response format. If the client sends an Accept: application/json header, the application returns a JSON response. If the client sends an Accept: application/xml header, the application returns an XML response.

Key points:

* **JSON and XML Support**: By specifying both JSON and XML as supported media types in your methods, you ensure that the application can dynamically respond with the correct format based on the client's request.

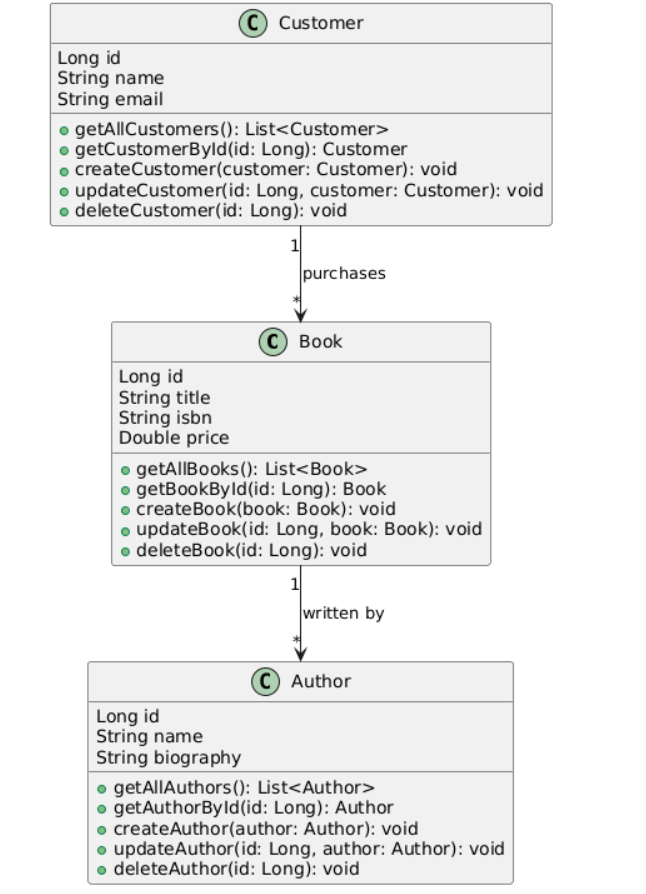
**FLOWCHART :**



**Explanation :**

1. **Start**: The process begins when a client sends an HTTP request.
2. **Check Request Header**: The controller checks the Accept header in the request to determine the preferred response format (either JSON or XML).
3. **Media Type Check**:
   * If the Accept header contains application/xml, the response is returned in XML format.
   * If the Accept header contains application/json (or any other format), the response is returned in JSON format.
4. **Process Request**: Based on the requested endpoint, the corresponding method in the controller (e.g., getAllBooks, getBookById) is invoked to retrieve or modify the data.
5. **Generate Response**: The controller generates the appropriate response with the data and the Content-Type header reflecting the chosen media type.
6. **End**: The response is sent back to the client.

**CLASS DIAGRAM :**



**Explanation of the Class Diagram**

1. **Book**: Represents the books in the bookstore. It contains attributes such as id, title, author, and price. The class has methods to get and set these attributes, and it is linked to the Order class through a many-to-one relationship (multiple books can be part of one order).
2. **Author**: Represents the authors of the books. It contains attributes such as id, name, and biography. The Author class has a one-to-many relationship with the Book class (one author can write multiple books).
3. **Customer**: Represents the customers of the bookstore. It contains attributes such as id, name, email, and address. The Customer class has a one-to-many relationship with the Order class (one customer can place multiple orders).
4. **Order**: Represents the orders placed by customers. It contains attributes such as id, orderDate, and status. The Order class is linked to both Book (many-to-one) and Customer (many-to-one), indicating that multiple books can be part of one order, and one customer can place multiple orders.