**Exercise 1: Online Bookstore - Setting Up RESTful Services**

1. Setup Spring Boot Project:

* Initialize a New Spring Boot Project:
  + Name: BookstoreAPI
  + Dependencies:
    - Spring Web (for building RESTful web services)
    - Spring Boot DevTools (for enhanced development features like hot-swapping)
    - Lombok (to reduce boilerplate code with annotations for getters, setters, etc.)
* Explanation:
  + This step involves creating a new Spring Boot project, which will serve as the foundation for the RESTful service. The selected dependencies are essential for developing a robust API with minimal boilerplate.

2. Project Structure:

* Generated Structure:
  + src/main/java: Contains the main application code.
  + src/main/resources: Stores configuration files (e.g., application.properties).
  + pom.xml: Lists the project’s dependencies and build configurations.
* Explanation:
  + Familiarizing yourself with the project structure is important to understand where to place different components (like controllers, services, and entities) in a Spring Boot application.

3. What's New in Spring Boot 3:

* Key New Features:
  + Java 17 Baseline: Requires Java 17, leveraging its new language features.
  + GraalVM Native Image Support: Allows building native executables with reduced startup time and memory usage.
  + Enhanced Observability: Improved metrics and tracing support through tools like Micrometer and OpenTelemetry.
  + HTTP/3 Support: Provides support for the latest HTTP protocol, enabling better performance.
  + Security Enhancements: Improved security defaults and better integration with Spring Security.
* Explanation:
  + Spring Boot 3 introduces several improvements that enhance performance, security, and observability, making it easier to build efficient and secure applications.

**Exercise 2: Online Bookstore - Creating Basic REST Controllers**

**1. Creating a Book Controller:**

* **BookController Class:** This is the class that will handle all incoming requests related to books. It is annotated with @RestController, which marks it as a controller where every method returns a domain object instead of a view. This is convenient for creating RESTful web services.
* **Request Mappings:** These are used to map web requests to specific handler functions in the controller. For example, you might map /books to a method that handles retrieving all books.

**2. Handling HTTP Methods:**

* **GET:** Used to retrieve information from the server. For example, GET /books might retrieve a list of all books.
* **POST:** Used to create a new book. The client sends a request body containing the book details, and the server saves it.
* **PUT:** Used to update an existing book. The client sends the updated book details in the request body.
* **DELETE:** Used to remove a book from the system based on its id.

**3. Returning JSON Responses:**

* **JSON Responses:** JSON (JavaScript Object Notation) is a lightweight data-interchange format. By default, Spring Boot returns JSON responses, so you only need to ensure that your methods return appropriate objects, and Spring Boot will handle the conversion.
* **Book Entity:** This is a simple Java class with attributes such as id, title, author, price, and isbn. This class represents the data structure that will be used in the application.

**Exercise 3: Online Bookstore - Handling Path Variables and Query Parameters**

**Business Scenario:**

Enhance the book management endpoints to handle dynamic URLs and query parameters.

**1. Path Variables:**

* **Definition:** Path variables are dynamic parts of the URL used to identify resources. For example, /books/{id} where {id} is a path variable representing the ID of a specific book.
* **Usage:** Implement an endpoint to fetch a book by its ID using this dynamic path variable. The server extracts the value of the ID from the URL and uses it to retrieve the corresponding book.

**2. Query Parameters:**

* **Definition:** Query parameters are key-value pairs appended to the end of a URL to filter or modify the results of an endpoint. For example, /books?title=Java&author=Smith would filter books with the title "Java" and author "Smith."
* **Usage:** Implement an endpoint that accepts query parameters to filter books based on attributes like title and author. The server processes these parameters and returns the matching books.

1. **Path Variables:**
   * The @GetMapping("/{id}") annotation maps the request to getBookById(@PathVariable int id) method, extracting the id from the URL to retrieve the specific book.
2. **Query Parameters:**
   * The getAllBooks method uses @RequestParam to accept optional query parameters title and author. These parameters are used to filter the list of books.
3. **Filtering Logic:**
   * The getAllBooks method filters the list of books using Java Streams. If the query parameters are not provided, it returns all books. If one or both parameters are provided, it returns books that match the criteria.

This implementation enhances the previous book management system by allowing more dynamic and flexible retrieval of books based on path variables and query parameters.

**Exercise 4: Online Bookstore - Processing Request Body and Form Data**

Business Scenario:

Create endpoints to accept and process JSON request bodies and form data for customer registrations.

**1. Request Body:**

* **Definition:** A request body in a POST request is the data sent by the client to the server to create or update a resource. In this scenario, the server will accept a JSON request body to create a new customer.
* **Usage:** Implement a POST endpoint where the client sends customer details in JSON format. The server will parse this JSON and use it to create a new customer entry in the system.

**2. Form Data:**

* **Definition:** Form data is typically sent when a user submits a form on a website. The data is sent as key-value pairs to the server, which processes the data to create or update a resource.
* **Usage:** Implement an endpoint that accepts form data for customer registrations. This is useful when processing data submitted through HTML forms.

1. **Request Body:**
   * The createCustomerFromJson method is annotated with @PostMapping("/json"), indicating that it will handle POST requests sent to /customers/json.
   * The @RequestBody annotation tells Spring to bind the incoming JSON request body to the Customer object. This object is then added to the customers list.
2. **Form Data:**
   * The createCustomerFromFormData method is annotated with @PostMapping("/form"), handling POST requests sent to /customers/form.
   * The @RequestParam annotations are used to bind individual form fields to method parameters. This method handles both text fields like name, email, password, and optional file uploads like profilePicture using MultipartFile.
3. **Handling Profile Picture:**
   * The profilePicture parameter is optional and processed if provided. In this example, the file is not saved, but you could implement file saving logic where necessary.

This implementation allows for versatile customer registration, accepting data through JSON request bodies and traditional form submissions.

**Exercise 5: Online Bookstore - Customizing Response Status and Headers**

**Business Scenario: Customize the HTTP response status and headers for the book management endpoints.**

**1. Response Status:**

* **Definition:** HTTP response status codes indicate the result of an HTTP request. For instance, 200 OK means the request was successful, while 404 Not Found indicates that the requested resource could not be found.
* **Usage:** You can use the @ResponseStatus annotation in Spring to customize the HTTP status code returned by a controller method. This allows you to communicate the result of the operation more clearly to the client.

**2. Custom Headers:**

* **Definition:** HTTP headers provide additional context or metadata about the request or response. Custom headers can be added to a response to convey extra information, such as the version of the API, the execution time of the request, etc.
* **Usage:** You can use ResponseEntity to build a response with custom headers and status codes. ResponseEntity is a powerful way to customize both the body of the response and its associated HTTP headers and status.

1. **Response Status:**
   * The @ResponseStatus annotation is used in the createBook method to automatically return an HTTP 201 (Created) status when a new book is successfully created.
   * The getBookById and deleteBook methods use ResponseEntity to customize the response status dynamically based on the outcome (e.g., 200 OK for success, 404 Not Found if the book does not exist).
2. **Custom Headers:**
   * In the getBookById and deleteBook methods, custom headers are added using HttpHeaders and passed into the ResponseEntity. For example, the header "Custom-Header": "Book-Fetch-Success" is added when a book is successfully retrieved.
   * These custom headers can be used by the client to gain additional information about the processing of the request.

This implementation gives you finer control over the HTTP response, allowing you to communicate more effectively with the client by using appropriate status codes and custom headers.

**Exercise 6: Online Bookstore - Exception Handling in REST Controllers**

**Business Scenario:**

Implement a global exception handling mechanism for the bookstore RESTful services.

**Global Exception Handler:**

* **Definition:** In a Spring application, exception handling can be centralized using a global exception handler. This allows you to manage exceptions across all RESTful services in one place rather than handling them individually in each controller.
* **Usage:** The @ControllerAdvice annotation is used to create a global exception handler class. Within this class, you define methods annotated with @ExceptionHandler to handle specific exceptions. These methods can return appropriate HTTP status codes and custom error messages.
* **Global Exception Handler:**
  + The GlobalExceptionHandler class is annotated with @ControllerAdvice, which makes it applicable across all controllers in the application.
  + The @ExceptionHandler methods are defined to handle specific exceptions like ResourceNotFoundException, IllegalArgumentException, and all other exceptions. Each method returns a ResponseEntity with a custom message and appropriate HTTP status code.
* **Custom Exceptions:**
  + The ResourceNotFoundException is a custom exception that extends RuntimeException. It is thrown in the BookController when a book with a specified ID is not found.
* **Controller Integration:**
  + The BookController is an example REST controller that uses the global exception handling. If a book with the specified ID is not found, a ResourceNotFoundException is thrown, and the global exception handler catches it, returning a 404 Not Found status with a custom message.

This setup ensures that all exceptions are handled in a consistent manner, improving the robustness and user experience of the API.

**Exercise 7: Online Bookstore - Introduction to Data Transfer Objects (DTOs)**

**Business Scenario:**

Use DTOs to transfer data between the client and server for books and customers.

**1. Data Transfer Objects (DTOs):**

* **Definition:** DTOs are simple objects used to transfer data between the client and server, or between different layers of an application. They contain only the necessary data fields required by the client and help decouple the internal domain model from the client.
* **Usage:** DTOs improve security by exposing only relevant data, reduce the amount of data sent over the network, and simplify data transformations.

**2. Mapping Entities to DTOs:**

* **Definition:** Mapping involves converting between entity classes (which typically represent the database model) and DTOs (which are used for communication with the client). This mapping can be done manually or with libraries like MapStruct or ModelMapper.
* **Usage:** Using a library simplifies the process and reduces boilerplate code, ensuring consistency and reducing the chances of errors in data conversion.

**3. Custom Serialization/Deserialization:**

* **Definition:** JSON serialization is the process of converting an object into a JSON string, while deserialization is the opposite process. Customizing this process allows you to control how your objects are converted to and from JSON.
* **Usage:** You can use Jackson annotations to customize JSON output, such as @JsonProperty for renaming fields, @JsonIgnore to exclude fields, and @JsonFormat to format dates.
* **DTO Classes:**
  + BookDTO and CustomerDTO are defined to transfer only the necessary data. For example, BookDTO includes only id, title, author, and price, with the price field renamed to cost using @JsonProperty.
  + CustomerDTO includes only the non-sensitive fields such as id, name, email, and registrationDate, with custom formatting for the date.
* **Mapping Entities to DTOs:**
  + The ModelMapper library is used to map entity objects (Book, Customer) to their respective DTOs (BookDTO, CustomerDTO). This simplifies the conversion process and ensures consistency.
  + In the BookstoreController, books and customers are mapped to their DTO counterparts before being returned as JSON responses.
* **Custom Serialization/Deserialization:**
  + Jackson annotations like @JsonProperty, @JsonFormat, and @JsonIgnore are used to customize the JSON serialization and deserialization process.
  + For example, the price field in BookDTO is serialized as cost, and the registrationDate in CustomerDTO is formatted as dd-MM-yyyy.

This approach ensures that only relevant data is exposed to the client, improving security, performance, and maintainability of the application.