Best practices for Controller, Service and Configuration

1. **Controllers:**

**Purpose:** Controllers are responsible for handling incoming HTTP requests and returning appropriate responses.

**Best Practices:**

1. The controller should be stateless! Controllers are singletons by default, and any state can cause a lot of problems.
2. The controller should not execute business logic, but rely on delegation.
3. The controller should handle the HTTP layer of the application, this should not be passed to the service.
4. Use @RestController for REST APIs: This annotation combines @Controller and @ResponseBody, simplifying the creation of REST endpoints.
5. **Use HTTP status codes appropriately:** Return the correct status codes (e.g., 200 OK, 400 Bad Request, 500 Internal Server Error).
6. **Validate input data:** Ensure data passed to controllers is valid before calling services.
7. **Handle exceptions gracefully:** Use @ControllerAdvice to handle global exceptions.
8. **Follow RESTful principles:** Use appropriate HTTP verbs (GET, POST, PUT, DELETE) and resource naming.
9. Use ResponseEntity for more control over responses: This allows you to set HTTP status codes, headers, and body.
10. **Use pagination and filtering:** Implement pagination and filtering to handle large datasets efficiently.
11. **Services:**

**Purpose:** Services encapsulate the business logic of your application.

**Best Practices:**

**Use interfaces and implementations:** Define service interfaces and implement them with concrete classes.

**Dependency Injection:** Use Spring's dependency injection to inject dependencies into services.

**Keep services thin:** Services should delegate complex tasks to other components or repositories.

**Use Spring Data JPA for database interactions:** Spring Data JPA simplifies data access and reduces boilerplate code.

**Handle exceptions within services:** Throw custom exceptions to be handled by controllers.

**Use logging:** Use SLF4J for logging.

**Use design patterns:** Apply relevant design patterns (e.g., Strategy, Template Method) to improve code organization and maintainability.

**Use services for business logic:** Do not put business logic inside controllers.

1. **Configuration:**

* **Purpose:** Configuration manages external settings and application properties.
* **Best Practices:**
* **Externalize configuration:** Avoid hardcoding configuration values in the code.
* **Use Spring Boot starters:** Leverage Spring Boot starters to automatically configure dependencies.
* **Use profiles for environment-specific configurations:** Define different configurations for different environments (e.g., development, testing, production).
* Use @ConfigurationProperties to bind properties to objects: This simplifies property management.
* Use @EnableConfigurationProperties to enable property binding: This annotation enables the binding of external properties to Java objects.
* Use @Value to inject values from properties: This allows you to inject values from properties into your beans.
* **Use Spring Boot Actuator:** Use Actuator to monitor and manage your application.
* **Use Spring Boot DevTools:** Use DevTools for faster development cycles.

**Use Spring Boot's built-in caching:** Use Spring's caching capabilities to improve performance.