Spring Boot Interview Questions & Answers

Chapter 11: Spring Boot – Exception Handling

Q: Why is exception handling important in Spring Boot applications?

Exception handling ensures that errors are caught gracefully and meaningful responses are sent to the client. Without it, users might see raw technical errors, which is unprofessional and insecure.

Use-Case / Example: In a banking API, if a user requests a non-existing account, instead of crashing, the system should return a proper error message like “Account not found”.

Q: How does Spring Boot handle exceptions by default?

By default, Spring Boot provides a BasicErrorController that returns a JSON response with details like timestamp, status, and error message whenever an exception occurs.

Use-Case / Example: If you hit an invalid endpoint /invalid-url, Spring Boot automatically returns a JSON error: { "timestamp": "...", "status": 404, "error": "Not Found" }

Q: What is @ExceptionHandler in Spring Boot?

@ExceptionHandler is used inside a controller to handle specific exceptions. It allows you to define custom responses for particular errors.

Use-Case / Example:

```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity handleNotFound(ResourceNotFoundException ex) {
return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
}
```

Q: What is @ControllerAdvice in Spring Boot?

@ControllerAdvice is a global error-handling mechanism that applies to all controllers. It centralizes exception handling in one place, making the code cleaner and easier to maintain.

Use-Case / Example: A global exception handler can return consistent error messages for all REST controllers, ensuring a uniform response format across the application.

Q: What is the difference between @ExceptionHandler and @ControllerAdvice?

- @ExceptionHandler handles exceptions only within the controller where it is defined.
- @ControllerAdvice applies globally across all controllers.

Use-Case / Example: If only UserController needs a custom handler for UserNotFoundException, use @ExceptionHandler. If the same exception can happen

across multiple controllers, use @ControllerAdvice.

Q: How can you customize error responses in Spring Boot?

You can create a custom error response object and return it inside your exception handler methods. This gives clients more structured and readable error messages.

Use-Case / Example:
class ErrorResponse {
private String message;
private int status;
private LocalDateTime timestamp;
}

Q: How do you handle validation errors in Spring Boot?

When using @Valid on request bodies, Spring Boot throws MethodArgumentNotValidException if validation fails. You can handle this in a global exception handler to return clear validation messages.

Use-Case / Example: If a signup form requires an email but the user enters an empty string, the API can return: { "error": "Email is required" }

Q: How do you return different HTTP status codes in exception handling?

You can use ResponseEntity to set custom HTTP status codes depending on the exception.

Use-Case / Example: For a missing resource, return 404 Not Found; for invalid input, return 400 Bad Request; for server issues, return 500 Internal Server Error.

Q: How does Spring Boot handle exceptions in REST APIs differently than MVC apps?

In REST APIs, Spring Boot usually returns JSON responses for errors, while in MVC apps, it returns error pages. Developers can override this behavior with custom handlers.

Use-Case / Example: In a REST API, a ProductNotFoundException returns a JSON error response, but in an MVC shopping website, it might return an HTML error page saying “Product not available”.

Q: Why is centralized exception handling a best practice?

Centralized handling improves maintainability, reduces duplication, and ensures consistent error responses across the application. It also makes debugging easier.

Use-Case / Example: A fintech company ensures that all microservices return errors in the same JSON format with fields like errorCode, message, and timestamp, making it easier for frontend and monitoring teams to consume.