1. What is Spring Boot and why is it used?

Spring Boot is a rapid application development framework built on top of Spring. It simplifies the process of setting up and running Spring applications by offering:

- Auto-configuration based on dependencies
- Embedded servers like Tomcat/Jetty
- Predefined starter templates (like spring-boot-starter-web)

This allows developers to build production-ready applications with minimal configuration.

2. What is the difference between @RestController and @Controller?

- @Controller is used in MVC applications to return views (like HTML pages).
- @RestController is a specialization of @Controller which returns data (typically JSON/XML) instead of views. It combines @Controller and @ResponseBody.

Use @RestController when building REST APIs.

3. What is the role of @RequestMapping?

@RequestMapping is a versatile annotation used to map web requests to specific handler methods in controller classes. It supports:

- Mapping different HTTP methods (GET, POST, etc.)
- URL patterns
- Request parameters and headers
- It provides full flexibility in defining routes but is now often replaced by specialized annotations.

4. What are @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping in Spring Boot?

These are specialized annotations that map HTTP requests to controller methods based on the HTTP method:

Annotation	HTTP Method	Use Case
`@GetMapping`	GET	Retrieve a resource
`@PostMapping`	POST	Create a new resource
`@PutMapping`	PUT	Update an existing resource
`@DeleteMapping`	DELETE	Delete a resource

They make the code more readable and aligned with REST principles.

5. What is the difference between @Component, @Service, and @Repository?

- @Component: Generic stereotype for Spring-managed components.
- @Service: Indicates a service layer class containing business logic.
- @Repository: Marks DAO (data access) classes and provides automatic exception translation.

6. What are the key features of REST architecture?

- **Stateless**: No session stored between client-server
- **Client-Server**: Loose coupling between frontend and backend
- **Cacheable**: Responses can be cached
- **Layered System**: API can work through intermediaries
- **Standard HTTP methods**: GET, POST, PUT, DELETE

7. What response is returned by Spring Boot if a requested resource is not found and no handler is defined?

Spring Boot returns a 404 Not Found status code along with a default error JSON indicating that the requested endpoint or path is unavailable.

8. What happens when a client sends malformed JSON or violates validation rules in a request body?

Spring Boot throws MethodArgumentNotValidException or HttpMessageNotReadableException, returning a 400 Bad Request status code with details of the validation errors in the response body.

9. What status code does Spring Boot return when a user tries to access a protected resource without authentication?

The API returns a 401 Unauthorized response, indicating that the request requires valid authentication credentials.

10. What is returned when an authenticated user accesses a resource they don't have permission for?

Spring returns a 403 Forbidden status code, meaning the server understood the request but refuses to authorize it for that user.

11. What does Spring Boot return when a POST request successfully creates a new resource?

The API responds with a 201 Created status and may include a Location header pointing to the newly created resource's URI.

12. In a Spring Boot REST application, what response does the client receive if an unhandled exception is thrown and no custom exception handling is implemented?

The client receives a default error response generated by Spring Boot, typically with a 500 Internal Server Error status code. The response body contains a standard JSON structure with details like timestamp, error message, status, and the exception trace (in development environments).

13. What status codes should be returned for different operations?

- 200 OK: Request succeeded (GET, PUT, DELETE)

- 201 Created: New resource created (POST)
- 204 No Content: Success, nothing to return
- 400 Bad Request: Invalid input or malformed request
- 401 Unauthorized: No valid authentication provided
- 403 Forbidden: Access denied
- 404 Not Found: Resource doesn't exist
- 500 Internal Server Error: Server-side issue

14. Difference between @RequestBody and @RequestParam?

- @RequestBody: Binds the entire request body to a Java object. Used for POST/PUT.
- @RequestParam: Extracts query parameters from the URL. Used mostly in GET requests.

15. What is the difference between @PathVariable and @RequestParam?

- @PathVariable: Binds a variable from the URL path (e.g., /user/{id})
- @RequestParam: Extracts values from query string (e.g., /user?id=123)

16. How to secure REST APIs in Spring Boot?

- Use **Spring Security** for authentication and authorization
- Integrate **JWT (JSON Web Token)** for stateless security
- Use annotations like @PreAuthorize, @Secured for role-based access control

17. How do you handle exceptions globally in Spring Boot REST API?

Use @ControllerAdvice along with @ExceptionHandler methods to catch and handle exceptions globally.

You can return a structured error response with timestamp, status, and error message.

18. What is the use of @ControllerAdvice and @ExceptionHandler?

- @ControllerAdvice: A global interceptor for all controllers, useful for centralized exception handling.
- @ExceptionHandler: Declares methods to handle specific exception types.

19. How can you set custom HTTP status codes in an exception handler?

- Use @ResponseStatus on custom exception classes
- Or return a ResponseEntity<Object> with custom status in the exception handler method

20. When is MethodArgumentNotValidException thrown in a Spring Boot REST application?

It is thrown when the request body fails to pass validation rules specified using annotations like @NotNull, @Size, etc., and the controller method is annotated with @Valid. This typically happens in @PostMapping or @PutMapping when invalid JSON is sent in the request.

21. How do you implement validation in Spring Boot (e.g., using @Valid)?

Use validation annotations in model (@NotNull, @Size) and annotate controller method with @Valid. Handle validation errors using BindingResult or @ExceptionHandler.

22. How would you design a REST API for a library management system?

- Endpoints: /books, /users, /borrow
- Use: POST to add, GET to fetch, PUT to update, DELETE to remove
- Use DTOs for input/output
- Validate inputs and handle exceptions

23. What is HATEOAS in REST?

HATEOAS (Hypermedia as the Engine of Application State) provides additional metadata (like links) in the response to help clients navigate the API.

24. What is the difference between @PutMapping and @PatchMapping in Spring REST, and when should each be used?

@PutMapping is used to replace the entire resource with the updated version. It expects the complete object in the request body.

@PatchMapping is used for partial updates, where only specific fields of a resource are updated, and the rest remain unchanged.

CODE BASED

1. In an e-commerce REST API, how would you implement an endpoint to create a new product and return a 201 Created response?

```
@PostMapping("/products")
public ResponseEntity<String> createProduct(@RequestBody Product product) {
   productService.save(product);
   return ResponseEntity.status(HttpStatus.CREATED).body("Product created successfully");
}
```

2. In a user management app, how would you validate user registration details using Spring validation annotations

```
@PostMapping("/register")
public ResponseEntity<String> register(@Valid @RequestBody User user) {
   userService.save(user);
   return ResponseEntity.ok("User registered");
}

public class User {
   @NotBlank
   private String username;

   @Email
   private String email;

   @Size(min = 6)
   private String password;
}
```

3. In an HR system, write a REST endpoint to fetch employee details by ID, and throw EmployeeNotFoundException if not found.

```
@GetMapping("/employees/{id}")
public ResponseEntity<Employee> getEmployee(@PathVariable Long id) {
   Employee employee = employeeRepository.findById(id)
      .orElseThrow(() -> new EmployeeNotFoundException("Employee not found"));
   return ResponseEntity.ok(employee);
}
```

4. In a hospital management system, how would you expose a REST endpoint to schedule a new appointment, ensuring the request body is validated?

```
@PostMapping("/appointments")
public ResponseEntity<String> bookAppointment(@Valid @RequestBody Appointment
appointment) {
   appointmentService.book(appointment);
   return ResponseEntity.status(HttpStatus.CREATED).body("Appointment booked");
}

public class Appointment {
   @NotNull
   private Long patientId;

   @Future
   private LocalDateTime appointmentDate;
}
```

5. In a library REST API, how would you implement a PATCH method to update only the book's availability status?

```
@PatchMapping("/books/{id}/availability")
public ResponseEntity<String> updateAvailability(@PathVariable int id, @RequestBody
boolean available) {
   bookService.updateAvailability(id, available);
   return ResponseEntity.ok("Availability updated");
}
```

6. In an HR management system, write a Spring REST controller method to remove an employee record. If the employee is not found, throw a custom EmployeeNotFoundException.

```
@DeleteMapping("/employees/{id}")
public ResponseEntity<String> removeEmployee(@PathVariable Long id) {
```

```
    if (!employeeRepository.existsById(id)) {
        throw new EmployeeNotFoundException("Employee not found with ID: " + id);
    }
    employeeRepository.deleteById(id);
    return ResponseEntity.ok("Employee deleted");
```

7. **Write code to globally handle UserNotFoundException and return a 404 status code.**

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }
}

public class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```