Top 5 Spring Boot Interview Questions to Ace in 2025

A Comprehensive Guide for Backend Developers

By Akash Singh

1. How does Spring Boots auto-configuration work?

Spring Boots **auto-configuration** simplifies setup by automatically configuring beans based on the projects classpath and dependencies. When you use `@SpringBootApplication` (which includes `@EnableAutoConfiguration`), Spring Boot scans the classpath for libraries like `spring-boot-starter-w` or `spring-boot-starter-data-jpa`. It then applies pre-defined configurations using `@ConditionalOnClass` (to check if a class is present) and `@ConditionalOnMissingBean` (to ensure user-defined beans take precedence). For example, if `spring-boot-starter-web` is detected, Spring Boot configures an embedded Tomcat server and `DispatcherServlet` for Spring MVC.

You can debug auto-configuration with the `-debug` flag or the `/actuator/conditions` endpoint in Spring Boot Actuator. To customize, exclude specific configurations:

Code Snippet

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.
    class})
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Tip: Understand how `spring.factories` defines auto-configuration classes for advanced customization.

Tricky Follow-Up Questions

- How does `@ConditionalOnClass` differ from `@ConditionalOnBean`?
- What happens if two auto-configurations conflict for the same bean?
- How would you create a custom auto-configuration for a library?

2. Whats the difference between @Controller and @RestController?

The `@Controller` annotation marks a class as a web controller that handles HTTP requests and typically returns view names (e.g., for Thymeleaf or JSP templates), resolved by a `ViewResolver`. In contrast, `@RestController` combines `@Controller` and `@ResponseBody`, returning data (e.g., JSON or XML) directly in the HTTP response, ideal for RESTful APIs in microservices. For example:

Code Snippet

```
@RestController
@RequestMapping("/api")
public class UserController {
    @GetMapping("/users")
    public List<User> getUsers() {
        return Arrays.asList(new User("Akash"), new User("John"));
    }
}
```

Tip: Use `@RestController` for API-driven microservices and `@Controller` for server-side rendered web applications.

Tricky Follow-Up Questions

- When would you use `@ResponseBody` with `@Controller` instead of `@RestController`?
- How does Spring Boot handle view resolution for `@Controller`?
- Can `@RestController` return a view instead of JSON?

3. How do you optimize Spring Boot startup time?

Optimizing startup time is critical for serverless environments, microservices, and developer productivity. Key techniques include:

- **Lazy Initialization**: Defer bean creation with `spring.main.lazy-initialization=true`, reducing startup time at the cost of a slower first request.
- **Spring Boot 3.x with Java 21s Features**: Use Java 21s virtual threads and GraalVMs Ahead-of-Time (AOT) compilation for native images, cutting startup by up to 50%.
- **Reduce Component Scanning**: Limit `@ComponentScan` to specific packages, e.g., `@ComponentScan(basePackages = "com.example.core")`.
- **Optimize Dependencies**: Remove unused starters and use `jdeps` to analyze classpath bloat.
- **Profile with Actuator**: Use `/actuator/startup` to identify slow beans.

Example for **lazy initialization**:

Code Snippet

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(Application.
            class);
        app.setLazyInitialization(true);
        app.run(args);
    }
}
```

Or in `application.properties`:

Code Snippet

```
spring.main.lazy-initialization=true
```

Tip: Highlight GraalVMs benefits for serverless and discuss trade-offs of **lazy initialization**.

Tricky Follow-Up Questions

- What are the risks of enabling **lazy initialization** in production?

– How does GraalVMs AOT compilation differ from JIT compilation?

– How would you debug a slow startup using Actuator metrics?

4. How do you handle global exceptions in Spring Boot?

Global exception handling ensures consistent error responses in REST APIs using **@ControllerAdvice** and `@ExceptionHandler`. This is crucial for microservices to standardize error handling across endpoints. For example, to handle `NullPointerException`:

Code Snippet

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(NullPointerException.class)
    public ResponseEntity<String> handleNPE(NullPointerException
        ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(
            "Invalid input: " + ex.getMessage());
    }
}
```

For custom exceptions:

Code Snippet

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

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

Tip: Structure error responses as JSON with error codes and messages for API consistency.

Tricky Follow-Up Questions

– How do you handle multiple exceptions in a single **@ControllerAdvice**?

– Can you customize error responses for different HTTP methods?

– How would you log exceptions globally for debugging?

5. How do you secure a Spring Boot REST API?

Securing a Spring Boot REST API involves adding `spring-boot-starter-security` and configuring a `SecurityFilterChain`. Modern APIs use OAuth2 or JWT for authentication. Additional considerations include enabling CORS for cross-origin requests, disabling CSRF for stateless APIs, and implementing rate-limiting to prevent abuse.

Example JWT configuration:

Code Snippet

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
        http) throws Exception {
        return http
            .cors(cors -> cors.configurationSource(request -> new
                CorsConfiguration().applyPermitDefaultValues()))
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated())
            .oauth2ResourceServer(oauth2 -> oauth2.jwt())
            .build();
    }

    @Bean
    public JwtDecoder jwtDecoder() {
        return NimbusJwtDecoder.withPublicKey(rsaPublicKey()).
            build();
    }
}
```

Add to `application.properties`:

Code Snippet

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://your-
    auth-server
```

For rate-limiting, use libraries like Resilience4j or a custom filter. *Tip*: Discuss OAuth2 vs. basic authentication and the importance of CORS/CSRF in microservices.

Tricky Follow-Up Questions

- How do you configure CORS for specific domains only?
- When should you enable CSRF in a Spring Boot API?
- How would you implement rate-limiting without external libraries?