10 Advanced Spring Boot Interview Questions That Separate Experts from Amateurs (With Code…

Most developers who have worked with Spring Boot think they're ready for interview questions on the topic. They've used dependency injection, written some REST controllers, and maybe even set up some simple JPA repositories. But when it comes to the tougher, more nuanced questions that separate truly knowledgeable developers from those who just know the basics, many fall short.

In this article, I'll walk you through some Spring Boot interview questions that most candidates aren't prepared for — questions that dig deeper into how Spring Boot actually works under the hood and how to solve real production problems.

1. "What's the difference between @SpringBootApplication, @EnableAutoConfiguration, and @ComponentScan?"

Many developers use `@SpringBootApplication` without understanding that it's actually a composite annotation that includes three annotations:

```
 @SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
```

The Standard Answer: Most candidates will say that `@SpringBootApplication` combines all three annotations as a convenience.

What They Should Say: A stronger answer breaks down what each does:

- `@SpringBootConfiguration` marks the class as a configuration class (it's a specialized form of `@Configuration`)
- `@EnableAutoConfiguration` tells Spring Boot to guess how you want to configure Spring based on the dependencies you've added
- `@ComponentScan` tells Spring to look for components in the current package and below

The Expert Answer: A true expert will also mention that you can exclude specific auto-configurations:

```
 @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
```

This is crucial when you want Spring Boot to stop automatically configuring certain beans that you want to define manually or don't need at all.

2. "How would you handle custom error pages in a Spring Boot application?"

The Standard Answer: Most candidates mention using static HTML error pages in the `/error` directory.

What They Should Say: A better answer includes implementing the `ErrorController` interface:

```
 @Controller
public class CustomErrorController implements ErrorController {

    @RequestMapping("/error")
    public String handleError(HttpServletRequest request) {
        Object status = request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
        if (status != null) {
            Integer statusCode = Integer.valueOf(status.toString());
            if (statusCode == HttpStatus.NOT_FOUND.value()) {
                return "error-404";
            } else if (statusCode == HttpStatus.INTERNAL_SERVER_ERROR.value()) {
                return "error-500";
            }
        }
        return "error";
    }
}
```

The Expert Answer: An expert would also mention the more elegant approach using `@ControllerAdvice`:

```
 @ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleResourceNotFoundException(ResourceNotFoundException ex) {
        ErrorResponse error = new ErrorResponse("NOT_FOUND", ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex) {
        ErrorResponse error = new ErrorResponse("INTERNAL_SERVER_ERROR", "An
unexpected error occurred");
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

They would explain that this approach allows for more fine-grained control over different exception types and formats of error responses.

3. "Explain Spring Boot Actuator and how you'd secure its endpoints in production"

The Standard Answer: Basic candidates will say Actuator provides production-ready features like health checks and metrics.

What They Should Say: A better answer includes specific endpoints and their purposes:

- `/health` shows application health information
- `/metrics` shows metrics information
- `/info` displays arbitrary application info
- `/env` shows environment variables

The Expert Answer: An expert will talk about securing these endpoints:

```
 @Configuration
public class ActuatorSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatcher(EndpointRequest.toAnyEndpoint())
            .authorizeRequests()
            .requestMatchers(EndpointRequest.to("health", "info")).permitAll()
            .anyRequest().hasRole("ACTUATOR_ADMIN")
            .and()
            .httpBasic();
    }
}
```

They'd also mention that you can:

1. Expose only specific endpoints: `management.endpoints.web.exposure.include=health,info`
2. Use different port for actuator endpoints: `management.server.port=8081`
3. Consider using Spring Security with JWT or OAuth2 for production environments

4. "How does Spring Boot's auto-configuration actually work?"

The Standard Answer: Most candidates will say something vague about "magic" or "scanning dependencies."

What They Should Say: A better answer explains that Spring Boot looks for JAR files that contain `META-INF/spring.factories` files which list auto-configuration classes.

The Expert Answer: A thorough answer includes:

1. Spring Boot scans for `META-INF/spring.factories` files in all JARs
2. These files list classes under the key `org.springframework.boot.autoconfigure.EnableAutoConfiguration`
3. Each auto-configuration class has `@Conditional` annotations that determine when it should be applied

4. Common conditions include:

- `@ConditionalOnClass` - only apply if a certain class is present
- `@ConditionalOnMissingBean` - only apply if a certain bean doesn't exist
- `@ConditionalOnProperty` - only apply if a certain property has a specific value

They would also explain that you can see which auto-configurations were applied and which were not by setting `debug=true` in your application properties.

5. "How would you implement custom validation in Spring Boot?"

The Standard Answer: Basic candidates mention using Bean Validation annotations like `@NotNull` and `@Size`.

What They Should Say: A better answer includes implementing a custom validator:

```
 @Constraint(validatedBy = AdultAgeValidator.class)
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface AdultAge {
    String message() default "Must be 18 or older";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
public class AdultAgeValidator implements ConstraintValidator<AdultAge, Integer> {
    @Override
    public boolean isValid(Integer age, ConstraintValidatorContext context) {
        return age != null && age >= 18;
    }
}
```

The Expert Answer: An expert would add handling validation errors with `@ControllerAdvice`:

```
 @ControllerAdvice
public class ValidationExceptionHandler {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public Map<String, String>
handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });
        return errors;
    }
}
```

They would also discuss group validation for different validation scenarios and mention programmatic validation using `Validator`.

6. "What's the difference between @Controller and @RestController?"

The Standard Answer: Most candidates will say that `@RestController` combines `@Controller` and `@ResponseBody`.

What They Should Say: A better answer explains that:

- `@Controller` is used in Spring MVC to return views
- `@RestController` automatically serializes return values to JSON/XML responses
- `@RestController` is preferred for RESTful web services

The Expert Answer: An expert will go deeper:

```
 // With @Controller, you need @ResponseBody for each method
@Controller
public class UserController {
    @GetMapping("/users")
    @ResponseBody
    public List<User> getUsers() {
        return userService.findAll();
    }

    @GetMapping("/users/view")
    public String getUsersView() {
        return "users-view"; // Returns a view name
    }
}
// With @RestController, everything is automatically treated as @ResponseBody
@RestController
public class UserApiController {
    @GetMapping("/api/users")
    public List<User> getUsers() {
        return userService.findAll(); // Automatically serialized to JSON
    }
}
```

They would also mention content negotiation, explaining that the client can request different formats (JSON, XML) using the Accept header, and Spring's message converters will handle the appropriate serialization.

7. "Explain Spring profiles and how you'd use them in a production environment"

The Standard Answer: Basic candidates will say profiles help manage different configurations for different environments.

What They Should Say: A better answer includes how to use profiles:

```
 @Configuration
@Profile("development")
public class DevelopmentConfig {
    // Development-specific beans
}
@Configuration
@Profile("production")
public class ProductionConfig {
    // Production-specific beans
}
```

The Expert Answer: An expert would discuss:

Profile-specific properties files:

- `application-dev.properties`
- `application-prod.properties`
- `application-test.properties`

Activating profiles:

- Via property: `spring.profiles.active=prod`
- Via environment variable: `SPRING_PROFILES_ACTIVE=prod`
- Programmatically: `SpringApplication.setAdditionalProfiles("prod")`

Profile groups (Spring Boot 2.4+):

```
spring.profiles.group.production=prod,metrics,actuator
```

Cloud-native approaches:

- Using Kubernetes ConfigMaps for environment-specific properties
- Using HashiCorp Vault for secrets management across environments
- Spring Cloud Config Server for centralized configuration

8. "How would you handle distributed transactions in a Spring Boot microservices architecture?"

The Standard Answer: Most candidates mention the challenge but don't know specific solutions.

What They Should Say: A stronger answer discusses the Saga pattern and event-driven approaches.

The Expert Answer: An expert answer includes several options:

Saga Pattern implementation:

```
 @Service
public class OrderSagaService {
    @Autowired private KafkaTemplate<String, OrderEvent> kafkaTemplate;

    @Transactional
    public void createOrder(Order order) {
        // Save order in PENDING state
        orderRepository.save(order);

        // Publish event for inventory service
        kafkaTemplate.send("inventory-events", new OrderEvent(order.getId(),
"CHECK_INVENTORY"));
    }

    @KafkaListener(topics = "inventory-responses")
    public void handleInventoryResponse(InventoryResponse response) {
        Order order =
orderRepository.findById(response.getOrderId()).orElseThrow();

        if (response.isAvailable()) {
            order.setStatus(OrderStatus.INVENTORY_CONFIRMED);
            orderRepository.save(order);
            // Trigger payment step
            kafkaTemplate.send("payment-events", new OrderEvent(order.getId(),
"PROCESS_PAYMENT"));
        } else {
            // Compensating transaction
            order.setStatus(OrderStatus.FAILED);
            orderRepository.save(order);
        }
    }
}
```

Mentioning frameworks that help with distributed transactions:

- Spring Cloud Stream for event-driven microservices
- Axon Framework for CQRS and Event Sourcing
- Eventuate Tram for implementing the Saga pattern

Discussing the CAP theorem and the tradeoffs between consistency and availability in distributed systems

9. "What are the pros and cons of using Spring Data JPA vs. direct Hibernate in a Spring Boot application?"

The Standard Answer: Basic answers mention that Spring Data JPA provides repositories and is easier to use.

What They Should Say: A more complete answer includes:

Pros of Spring Data JPA:

- Reduces boilerplate with built-in repository methods
- Automatic implementation of query methods based on method names
- Pagination and sorting support
- Auditing capabilities

Cons:

- Abstraction can hide important Hibernate details
- Potentially complex queries may be harder to express
- Performance tuning might require dropping down to native SQL

The Expert Answer: An expert would provide specific examples:

```
 // With Spring Data JPA:
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByEmailContainingOrderByLastNameAsc(String email);

    @Query("SELECT u FROM User u WHERE u.status = :status AND u.lastLogin >
:date")
    List<User> findActiveUsersSince(@Param("status") Status status, @Param("date")
LocalDate date);
}
// With direct Hibernate:
@Repository
public class UserDao {
    @PersistenceContext
    private EntityManager entityManager;

    public List<User> findActiveUsersSince(Status status, LocalDate date) {
        Session session = entityManager.unwrap(Session.class);

        CriteriaBuilder cb = session.getCriteriaBuilder();
        CriteriaQuery<User> query = cb.createQuery(User.class);
        Root<User> root = query.from(User.class);

        query.select(root)
             .where(cb.and(
                 cb.equal(root.get("status"), status),
                 cb.greaterThan(root.get("lastLogin"), date)
             ));

        return session.createQuery(query).getResultList();
    }
}
```

They would also discuss:

1. How to optimize N+1 query problems in Spring Data JPA
2. When to use `@EntityGraph` for fetching related entities
3. The importance of understanding the persistence context and transaction boundaries

10. "How would you implement rate limiting in a Spring Boot API?"

The Standard Answer: Most candidates mention using a third-party API gateway.

What They Should Say: A better answer includes implementing it within the application:

```
 @Configuration
public class RateLimitConfig {
    @Bean
    public RateLimiter userRateLimiter() {
        return RateLimiter.create(10.0); // 10 requests per second
    }
}
@RestController
public class UserController {
    @Autowired
    private RateLimiter userRateLimiter;

    @GetMapping("/api/users")
    public ResponseEntity<List<User>> getUsers() {
        if (!userRateLimiter.tryAcquire()) {
            return ResponseEntity.status(HttpStatus.TOO_MANY_REQUESTS).build();
        }
        return ResponseEntity.ok(userService.findAll());
    }
}
```

The Expert Answer: An expert would implement a more sophisticated solution with a custom annotation:

```
 @Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface RateLimit {
    int value();           // requests per minute
    String key() default "";  // rate limit key, e.g. by user, by IP
}
@Aspect
@Component
public class RateLimitAspect {
    private Map<String, Bucket> buckets = new ConcurrentHashMap<>();

    @Around("@annotation(rateLimit)")
    public Object enforceRateLimit(ProceedingJoinPoint joinPoint, RateLimit
rateLimit) throws Throwable {
        HttpServletRequest request = ((ServletRequestAttributes)
RequestContextHolder.getRequestAttributes()).getRequest();

        String key = rateLimit.key().isEmpty() ?
            request.getRemoteAddr() : ReflectionUtils.invokeMethod(
                ReflectionUtils.findMethod(request.getClass(), "get" +
StringUtils.capitalize(rateLimit.key())),
                request
            ).toString();

        Bucket bucket = buckets.computeIfAbsent(key,
            k -> Bucket4j.builder().addLimit(Bandwidth.simple(rateLimit.value(),
Duration.ofMinutes(1))).build());

        if (bucket.tryConsume(1)) {
            return joinPoint.proceed();
        } else {
            throw new TooManyRequestsException("Rate limit exceeded");
        }
    }
}
@RestController
public class ProductController {
    @GetMapping("/api/products")
    @RateLimit(value = 30)  // 30 requests per minute by IP
    public List<Product> getProducts() {
        return productService.findAll();
    }

    @GetMapping("/api/users/{userId}/orders")
    @RateLimit(value = 10, key = "userId")  // 10 requests per minute by user ID
    public List<Order> getUserOrders(@PathVariable String userId) {
        return orderService.findByUserId(userId);
    }
}
```

They would also discuss distributed rate limiting using Redis to handle multiple application instances.

Conclusion

The difference between a good Spring Boot developer and a great one often comes down to understanding what's happening under the hood. By preparing for these more advanced questions, you'll not only ace your interview but also become a more effective developer who can solve real-world problems.

In the comments, I'd love to hear about the toughest Spring Boot interview questions you've faced. What questions do you think should be on this list?

Follow me for more articles on Spring, Java, and backend development best practices. If you found this helpful, consider sharing it with your network!