Spring Boot & Spring Framework Annotations Guide

A comprehensive reference document for your Spring Boot interview preparation. This guide covers all important annotations in the Spring ecosystem, explaining their purpose and usage.

Table of Contents

Core Spring Annotations

These annotations form the backbone of the Spring dependency injection and bean management system.

Bean Configuration

`@Bean`

- Purpose: Marks a method as a bean producer
- Usage: Used in @Configuration classes to explicitly declare beans
- Example:

```
@Bean
public UserService userService() {
    return new UserServiceImpl();
}
```

`@Configuration`

- Purpose: Indicates a class defines Spring beans
- Usage: Primary class for defining beans via @Bean methods
- Notes: Processed during application startup to register beans

`@ComponentScan`

- Purpose: Tells Spring where to look for components
- Usage: Automatically scans specified packages for Spring components
- Example:

```
@Configuration
@ComponentScan("com.example.app")
public class AppConfig {}
```

Component Annotations

`@Component`

- Purpose: Generic stereotype for Spring-managed components
- Usage: Base annotation indicating a class is a Spring component
- Notes: Parent annotation for @Service, @Repository, and @Controller

`@Controller`

- Purpose: Marks a class as Spring MVC controller
- Usage: Web controllers that handle HTTP requests
- Notes: Specialized form of @Component

`@RestController`

- Purpose: Combines @Controller and @ResponseBody
- Usage: For RESTful web services where all methods return domain objects
- Notes: Eliminates need for @ResponseBody on each method

`@Service`

- Purpose: Indicates a business service component
- Usage: For classes implementing business logic
- Notes: Functionally same as @Component but more semantically meaningful

`@Repository`

- Purpose: Indicates a data access component
- Usage: For DAO classes dealing with database operations
- Notes: Enables automatic persistence exception translation

Dependency Injection

`@Autowired`

- Purpose: Marks a dependency for injection
- Usage: Used on fields, constructors, or methods for automatic wiring
- Example:

```
@Autowired
private UserService userService;
```

`@Qualifier`

- Purpose: Specifies which bean to inject when multiple beans of same type exist
- Usage: Used with @Autowired to specify a specific bean by name
- Example:

```
@Autowired
@Qualifier("primaryUserService")
private UserService userService;
```

@Primary

- Purpose: Designates a bean as the primary candidate for autowiring
- Usage: When multiple beans of the same type exist, the @Primary one is chosen
- Example:

```
@Service
@Primary
public class PrimaryUserService implements UserService {}
```

@Value

- Purpose: Injects values from properties files or expressions
- Usage: Injects property values into beans
- Example:

```
@Value("${app.timeout}")
private int timeout;
```

@Lazy

- Purpose: Delays bean initialization until first use
- Usage: Makes a bean lazy-initialized
- Notes: Can improve startup time but may shift errors to runtime

@Required

- Purpose: Indicates a required property must be populated
- Usage: Used on setter methods
- Notes: Deprecated in Spring 5.1, replaced by constructor injection or @Autowired with required=true

@Scope

- Purpose: Defines the scope of a bean
- Usage: Controls bean lifecycle and visibility
- Example:

```
```

```
@Bean
@Scope("prototype")
public UserService userService() {
    return new UserServiceImpl();
}
```

- Common scopes: singleton (default), prototype, request, session, application

`@DependsOn`

- Purpose: Forces specific beans to initialize before this one
- Usage: Ensures dependency initialization order
- Example:

```
@Bean
@DependsOn("dataSource")
public UserRepository userRepository() {
    return new JdbcUserRepository();
}
```

Aspect-Oriented Programming

`@Aspect`

- Purpose: Marks a class as an aspect
- Usage: Used with @Component to define cross-cutting concerns
- Notes: Requires Spring AOP or AspectJ

`@Pointcut`

- Purpose: Defines a reusable pointcut expression
- Usage: Specifies where advice should be applied
- Example:

```
@Pointcut("execution(* com.example.service.*.*(..))")
public void serviceMethods() {}
```

`@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`, `@Around`

- Purpose: Define advice to execute at specific join points
- Usage: Apply cross-cutting behavior
- Example:

```
@Before("serviceMethods()")
public void logMethodCall(JoinPoint jp) {
```

```
        logger.info("Executing: " + jp.getSignature());
    }
```

Spring Boot Annotations

Annotations specific to Spring Boot applications, focusing on auto-configuration and application setup.

@SpringBootApplication

- Purpose: Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan
- Usage: Primary annotation for Spring Boot applications
- Example:

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

@EnableAutoConfiguration

- Purpose: Enables Spring Boot's auto-configuration mechanism
- Usage: Automatically configures beans based on classpath
- Notes: Usually not used directly (part of @SpringBootApplication)

@ConfigurationProperties

- Purpose: Binds properties to configuration classes
- Usage: Type-safe configuration from properties files
- Example:

```
@ConfigurationProperties(prefix = "app.mail")
public class MailProperties {
    private String host;
    private int port;
    // getters and setters
}
```

@EnableConfigurationProperties

- Purpose: Enables support for @ConfigurationProperties classes
- Usage: Register @ConfigurationProperties beans
- Example:

```

```

```
@Configuration
@EnableConfigurationProperties(MailProperties.class)
public class MailConfig {}
```

`@ConditionalOnBean`, `@ConditionalOnMissingBean`

- Purpose: Conditionally include beans based on presence/absence of other beans
- Usage: For conditional auto-configuration
- Example:

```
@Bean
@ConditionalOnMissingBean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder().build();
}
```

`@ConditionalOnProperty`

- Purpose: Conditionally include beans based on property values
- Usage: For property-driven configuration
- Example:

```
@Bean
@ConditionalOnProperty(name = "app.feature.enabled", havingValue = "true")
public FeatureService featureService() {
    return new FeatureServiceImpl();
}
```

`@ConditionalOnClass`, `@ConditionalOnMissingClass`

- Purpose: Conditionally include beans based on class presence
- Usage: For classpath-dependent configuration
- Example:

```
@Bean
@ConditionalOnClass(name =
"org.springframework.data.redis.core.RedisTemplate")
public CacheManager cacheManager() {
    return new RedisCacheManager();
}
```

`@ConditionalOnWebApplication`, `@ConditionalOnNotWebApplication`

- Purpose: Conditionally include beans based on application type
- Usage: For web/non-web specific configuration

`@ImportAutoConfiguration`

- Purpose: Imports specific auto-configuration classes
- Usage: For targeted auto-configuration loading

`@AutoConfigureAfter`, `@AutoConfigureBefore`, `@AutoConfigureOrder`

- Purpose: Control auto-configuration order
- Usage: For fine-tuning auto-configuration loading sequence

Spring Web MVC Annotations

Annotations for building web applications and REST services with Spring MVC.

`@RequestMapping`

- Purpose: Maps web requests to handler methods
- Usage: Defines URL patterns handled by controller
- Example:

```
@RequestMapping("/users")
public class UserController {}
```

`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@PatchMapping`

- Purpose: Specialized versions of @RequestMapping for specific HTTP methods
- Usage: More concise way to define method-specific endpoints
- Example:

```
@GetMapping("/{id}")
public User getUser(@PathVariable Long id) {}
```

`@RequestParam`

- Purpose: Binds request parameters to method parameters
- Usage: For accessing query parameters and form data
- Example:

```
@GetMapping("/search")
public List<User> search(@RequestParam String name) {}
```

`@PathVariable`

- Purpose: Binds URI template variables to method parameters
- Usage: For dynamic parts of the URL
- Example:

```
@GetMapping("/{id}")
public User getUser(@PathVariable("id") Long userId) {}
```

`@RequestBody`

- Purpose: Binds HTTP request body to a method parameter
- Usage: For JSON/XML payloads in requests
- Example:

```
@PostMapping
public User createUser(@RequestBody User user) {}
```

`@ResponseBody`

- Purpose: Indicates return value should be bound to web response body
- Usage: For returning objects as response body (JSON/XML)
- Notes: Not needed with @RestController

`@RequestHeader`

- Purpose: Binds request header values to method parameters
- Usage: For accessing HTTP headers
- Example:

```
@GetMapping
public String handleRequest(@RequestHeader("User-Agent") String userAgent)
{}
```

`@CookieValue`

- Purpose: Binds cookie values to method parameters
- Usage: For accessing cookie data
- Example:

```
@GetMapping
public String handle(@CookieValue("sessionId") String sessionId) {}
```

`@SessionAttribute`, `@SessionAttributes`

- Purpose: Binds session attributes to method parameters
- Usage: For controlling session scope attributes
- Example:

```
@SessionAttributes("user")
public class UserController {}
```

`@ModelAttribute`

- Purpose: Binds form data to model objects
- Usage: Pre-populates model or binds form submissions
- Example:

```
@PostMapping
public String submitForm(@ModelAttribute User user) {}
```

`@ExceptionHandler`

- Purpose: Handles exceptions thrown by controller methods
- Usage: For controller-specific error handling
- Example:

```
@ExceptionHandler(UserNotFoundException.class)
public ResponseEntity<Error> handleNotFound(UserNotFoundException ex) {}
```

`@ControllerAdvice`, `@RestControllerAdvice`

- Purpose: Applies @ExceptionHandler and other annotations globally
- Usage: For global error handling and model enhancements
- Example:

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Error> handleException(Exception ex) {}
}
```

`@ResponseStatus`

- Purpose: Sets HTTP status code for method response
- Usage: For custom HTTP status responses
- Example:

```
@ResponseStatus(HttpStatus.CREATED)
@PostMapping
public void createUser(@RequestBody User user) {}
```

`@CrossOrigin`

- Purpose: Enables cross-origin resource sharing (CORS)
- Usage: For controlling CORS behavior
- Example:

```
@CrossOrigin(origins = "http://example.com")
@GetMapping("/api/users")
public List<User> getUsers() {}
```

Spring Data Annotations

Annotations for database access and object-relational mapping.

`@Entity`

- Purpose: Marks a class as JPA entity
- Usage: For classes mapped to database tables
- Example:

```
@Entity
public class User {
    @Id
    private Long id;
}
```

`@Table`

- Purpose: Specifies the table name for an entity
- Usage: When entity name differs from table name
- Example:

```
@Entity
@Table(name = "users")
public class User {}
```

`@Id`

- Purpose: Marks a field as the primary key
- Usage: Identifies primary key field in entities
- Notes: Required for every JPA entity

`@GeneratedValue`

- Purpose: Specifies how the primary key should be generated
- Usage: For auto-generation of primary keys

- Example:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

`@Column`

- Purpose: Specifies column mapping for a field
- Usage: For customizing column properties
- Example:

```
@Column(name = "user_name", length = 50, nullable = false)
private String username;
```

`@JoinColumn`, `@JoinTable`

- Purpose: Specifies relationship mapping
- Usage: For defining foreign key columns and join tables
- Example:

```
@OneToMany
@JoinColumn(name = "user_id")
private List<Order> orders;
```

`@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`

- Purpose: Defines entity relationships
- Usage: For mapping relationships between entities
- Example:

```
@ManyToOne
private Department department;
```

`@Transient`

- Purpose: Marks a field as not persistent
- Usage: For fields not stored in database
- Example:

```
@Transient
private int calculatedValue;
```

@Temporal

- Purpose: Specifies date/time field type
- Usage: For date/time precision control
- Example:

```
@Temporal(TemporalType.DATE)
private Date birthDate;
```

@Query

- Purpose: Defines custom JPQL or SQL queries
- Usage: For complex queries in repositories
- Example:

```
@Query("SELECT u FROM User u WHERE u.status = ?1")
List<User> findByStatus(String status);
```

@Modifying

- Purpose: Indicates a query method performs DML operation
- Usage: For update and delete queries
- Example:

```
@Modifying
@Query("UPDATE User u SET u.status = ?2 WHERE u.id = ?1")
int updateStatus(Long id, String status);
```

@Transactional

- Purpose: Declares transaction boundaries
- Usage: For transactional methods or classes
- Example:

```
@Transactional
public void transferFunds(Account from, Account to, BigDecimal amount) {}
```

@CreatedDate, @LastModifiedDate, @CreatedBy, @LastModifiedBy

- Purpose: Auditing annotations
- Usage: For tracking entity creation and modification
- Notes: Used with @EnableJpaAuditing

@Version

- Purpose: Enables optimistic locking
- Usage: For concurrency control
- Example:

```
@Version
private Long version;
```

Spring Security Annotations

Annotations for securing Spring applications.

`@EnableWebSecurity`

- Purpose: Enables Spring Security's web security support
- Usage: In security configuration classes
- Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {}
```

`@EnableGlobalMethodSecurity`

- Purpose: Enables method-level security
- Usage: For securing methods with annotations
- Example:

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig {}
```

`@PreAuthorize`, `@PostAuthorize`

- Purpose: Defines access control expressions for methods
- Usage: For method security
- Example:

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser(Long id) {}
```

`@Secured`

- Purpose: Secures methods with static role expressions
- Usage: Simpler alternative to @PreAuthorize

- Example:

```
@Secured("ROLE_ADMIN")
public void updateUser(User user) {}
```

`@RolesAllowed`

- Purpose: Java standard annotation for role-based security
- Usage: Equivalent to @Secured
- Example:

```
@RolesAllowed("ADMIN")
public void resetDatabase() {}
```

`@AuthenticationPrincipal`

- Purpose: Injects authenticated principal
- Usage: For accessing current user in controllers
- Example:

```
@GetMapping("/profile")
public User getProfile(@AuthenticationPrincipal UserDetails user) {}
```

Spring Test Annotations

Annotations for testing Spring applications.

`@RunWith(SpringRunner.class)`

- Purpose: Integrates JUnit with Spring
- Usage: For Spring-powered JUnit 4 tests
- Notes: Use @ExtendWith in JUnit 5

`@ExtendWith(SpringExtension.class)`

- Purpose: JUnit 5 equivalent of @RunWith(SpringRunner.class)
- Usage: For Spring-powered JUnit 5 tests

`@SpringBootTest`

- Purpose: Sets up full Spring Boot test context
- Usage: For integration tests with Spring Boot
- Example:

```

```

```
@SpringBootTest
public class UserServiceIntegrationTest {}
```

@WebMvcTest

- Purpose: Tests Spring MVC controllers
- Usage: For controller-layer tests
- Example:

```
@WebMvcTest(UserController.class)
public class UserControllerTest {}
```

@DataJpaTest

- Purpose: Tests JPA repositories
- Usage: For repository-layer tests
- Example:

```
@DataJpaTest
public class UserRepositoryTest {}
```

@MockBean

- Purpose: Creates and registers a Mockito mock for a bean
- Usage: For mocking dependencies in tests
- Example:

```
@MockBean
private UserService userService;
```

@SpyBean

- Purpose: Creates and registers a Mockito spy for a bean
- Usage: For partially mocking beans in tests

@ActiveProfiles

- Purpose: Sets active profiles for tests
- Usage: For profile-specific test configuration
- Example:

```
@SpringBootTest
@ActiveProfiles("test")
```

```
public class ApplicationTest {}
```

`@Sql`

- Purpose: Executes SQL scripts before/after tests
- Usage: For database setup in tests
- Example:

```
@Test
@Sql("/test-data.sql")
public void testWithDatabaseSetup() {}
```

Spring Cloud Annotations

Annotations for building distributed systems with Spring Cloud.

`@EnableDiscoveryClient`

- Purpose: Registers with service discovery server
- Usage: For service registry integration
- Notes: Works with any service discovery implementation

`@EnableEurekaClient`

- Purpose: Registers with Eureka server
- Usage: For Netflix Eureka-specific integration
- Notes: More specific than @EnableDiscoveryClient

`@EnableCircuitBreaker`

- Purpose: Enables circuit breaker pattern
- Usage: For fault tolerance
- Notes: Used with Hystrix or Resilience4j

`@HystrixCommand`

- Purpose: Applies circuit breaker to method
- Usage: For method-level fault tolerance
- Example:

```
@HystrixCommand(fallbackMethod = "getDefaultUser")
public User getUser(Long id) {}
```

`@EnableFeignClients`

- Purpose: Enables Feign clients
- Usage: For declarative REST clients

- Example:

```
@SpringBootApplication
@EnableFeignClients
public class Application {}
```

`@FeignClient`

- Purpose: Declares a Feign client interface
- Usage: For REST API clients
- Example:

```
@FeignClient(name = "user-service")
public interface UserClient {
    @GetMapping("/users/{id}")
    User getUser(@PathVariable Long id);
}
```

`@EnableZuulProxy`

- Purpose: Enables Zuul API gateway
- Usage: For API routing and filtering
- Notes: Specifically for Netflix Zuul

`@EnableConfigServer`

- Purpose: Enables Spring Cloud Config Server
- Usage: For centralized configuration

`@EnableConfigClient`

- Purpose: Enables Spring Cloud Config Client
- Usage: For obtaining configuration from Config Server
- Notes: Not required in newer versions

Other Important Annotations

Additional annotations that don't fit into the above categories.

`@Scheduled`

- Purpose: Schedules method execution
- Usage: For periodic tasks
- Example:

```
@Scheduled(fixedRate = 5000)
public void generateReport() {}
```

`@Async`

- Purpose: Marks a method to run asynchronously
- Usage: For non-blocking operations
- Example:

```
@Async
public CompletableFuture<User> findUserAsync(Long id) {}
```

`@EnableAsync`

- Purpose: Enables asynchronous method execution
- Usage: Required for @Async to work
- Example:

```
@Configuration
@EnableAsync
public class AsyncConfig {}
```

`@EnableCaching`

- Purpose: Enables Spring's caching infrastructure
- Usage: Required for caching annotations to work

`@Cacheable`, `@CachePut`, `@CacheEvict`

- Purpose: Controls method-level caching
- Usage: For caching method results
- Example:

```
@Cacheable("users")
public User getUser(Long id) {}
```

`@EnableScheduling`

- Purpose: Enables scheduling capabilities
- Usage: Required for @Scheduled to work
- Example:

```
@Configuration
@EnableScheduling
public class SchedulingConfig {}
```

@EnableRetry

- Purpose: Enables Spring Retry
- Usage: For automatic retrying of failed operations

@Retryable

- Purpose: Marks methods for retry
- Usage: For operations that may temporarily fail
- Example:

```
@Retryable(maxAttempts = 3, backoff = @Backoff(delay = 1000))
public void unreliableOperation() {}
```

@Profile

- Purpose: Conditionally enables beans based on active profiles
- Usage: For environment-specific configuration
- Example:

```
@Bean
@Profile("development")
public DataSource developmentDataSource() {}
```

@PropertySource

- Purpose: Imports property files
- Usage: For loading additional property files
- Example:

```
@Configuration
@PropertySource("classpath:custom.properties")
public class AppConfig {}
```

@Order

- Purpose: Defines ordering of annotated components
- Usage: For controlling initialization order
- Example:

```
@Component
@Order(1)
public class HighPriorityComponent {}
```

`@EventListener`

- Purpose: Marks methods as event listeners
- Usage: For application event handling
- Example:

```
@EventListener
public void handleUserCreatedEvent(UserCreatedEvent event) {}
```

Spring Interview Tips

Beyond annotations, these topics are often covered in Spring Boot interviews:

1. **Dependency Injection Principles** - Understand the core concepts behind Spring's IoC container
2. **Bean Lifecycle** - Initialization, destruction, callback methods, and bean scopes
3. **Aspect-Oriented Programming** - Uses, benefits, and implementation details
4. **Spring Boot Auto-Configuration** - How it works and how to customize it
5. **Spring Profiles** - Using profiles for environment-specific configuration
6. **Spring Data Repository Patterns** - JpaRepository, CrudRepository, pagination, etc.
7. **Transaction Management** - Declarative vs. programmatic, isolation levels, propagation
8. **Spring Security Architecture** - Authentication, authorization, filters
9. **Testing Strategies** - Unit tests, integration tests, and Spring's testing support
10. **Microservices with Spring Cloud** - Service discovery, configuration, routing

Remember to relate your answers to practical scenarios and projects you've worked on. Good luck with your interview!