30 Spring Boot Annotations

Every Interviewer Expects You to Know

With Examples, Explanations & Pitfalls

@SpringBootApplication

- **Meta-annotation: combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.**
- **Place on the root package class so component scanning covers subpackages.**
- **Bootstraps app with sensible defaults via auto-configuration.**

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

Note: Avoid placing it in a nested package; components outside scan base won ’ t load.

@Configuration

- **Designates a class as a source of bean definitions.**
- **Pairs with @Bean methods to declare Spring-managed objects.**
- **Often used to customize or override auto-configured beans.**

```
@Configuration
public class MetricsConfig {
 @Bean
 public MeterRegistry meterRegistry() { return new SimpleMeterRegistry(); }
}
```

@EnableAutoConfiguration

- **Lets Spring Boot configure beans based on classpath and properties.**
- **Use 'exclude' to turn off specific auto-configurations when needed.**

```
@EnableAutoConfiguration(exclude = { DataSourceAutoConfiguration.class })
public class CustomApp { }
```

Note: Included automatically by @SpringBootApplication.

@ComponentScan

- **Controls which packages are scanned for stereotypes (@Component, @Service, etc.).**
- **Defaults to the package of the config class; customize with basePackages/basePackageClasses.**

```
@ComponentScan(basePackages = {"com.example.api","com.example.core"})
public class ScanConfig { }
```

Note: Keep scan scope minimal to reduce startup time & collisions.

@Bean

- **Declares a bean method; return value is registered in the application context.**
- **Good for third-party types or objects not created via component scanning.**
- **Default bean name is method name; can be overridden.**

```
@Configuration
public class AppConfig {
 @Bean(name="objectMapper")
 public ObjectMapper mapper() { return new ObjectMapper(); }
}
```

Note: Singleton by default; change scope with @Scope.

@Component

- Generic stereotype for a Spring-managed bean.
- Used when more specific stereotypes do not fit.

```
@Component
public class IdGenerator {
  public String next() { return UUID.randomUUID().toString(); }
}
```

@Service

- **Stereotype for service-layer components (business logic).**
- **Semantics improve readability; commonly paired with @Transactional.**

```
@Service
public class OrderService {
 public void placeOrder(CreateOrder req) { /* ... */ }
}
```

@Repository

- **Stereotype for persistence/DAO layer components.**
- **Enables exception translation to DataAccessException hierarchy.**

```
@Repository
public class UserRepositoryCustom {
  // custom JDBC/JPA operations
}
```

Note: Spring Data interfaces (e.g., JpaRepository) are already treated as repositories.

@Controller

- Marks a Spring MVC controller that returns view names/templates.
- Use when rendering HTML (e.g., Thymeleaf, Freemarker).

```
@Controller
public class PageController {
 @GetMapping("/home")
 public String home() { return "home"; }
}
```

Note: For JSON responses, add @ResponseBody or switch to @RestController.

@RestController

- Combination of @Controller + @ResponseBody; returns JSON/XML directly.
- Ideal for REST APIs; no view resolution.

```
@RestController
@RequestMapping("/api/users")
public class UserApi {
  @GetMapping("/{id}")
  public UserDto find(@PathVariable Long id) { return service.find(id); }
}
```

@Autowired

- **Injects dependencies by type; prefer constructor injection.**
- **Use 'required=false' for optional dependencies (or Optional).**

```
@Service
public class BillingService {
  private final PaymentGateway gateway;
  @Autowired
  public BillingService(PaymentGateway gateway) { this.gateway = gateway; }
}
```

Note: With a single constructor, @Autowired is optional in recent Spring versions.

@Qualifier

- **Resolves ambiguity when multiple beans of the same type exist.**
- **Use on fields/constructors/parameters to specify bean name.**

```
@Bean("primaryGateway") PaymentGateway a(){ return new StripeGateway(); }
@Bean("backupGateway")  PaymentGateway b(){ return new RazorpayGateway(); }

@Service
class BillingService {
  BillingService(@Qualifier("primaryGateway") PaymentGateway gw){ /*...*/ }
}
```

@Primary

- Marks a bean as the default candidate for autowiring when conflicts exist.
- Can be overridden by an explicit @Qualifier at injection site.

```
@Primary
@Bean
PaymentGateway mainGw(){ return new StripeGateway(); }
```

@Value

- Injects simple values or SpEL from properties/environment.
- Prefer @ConfigurationProperties for structured configs.

```
@Value("${app.region:us-east-1}")
private String region;
```

@ConfigurationProperties

- **Binds external configuration to a POJO under a prefix.**
- **Supports validation when combined with @Validated or field-level constraints.**

```
@ConfigurationProperties(prefix="payment")
public class PaymentProps {
  private String provider;
  private int timeoutMs;
  // getters/setters
}
```

Note: Register via @ConfigurationPropertiesScan or @EnableConfigurationProperties, or annotate a @Component.

@RequestMapping / @GetMapping / @PostMapping

- **Maps paths and HTTP attributes to handler methods.**
- **Composed variants (@GetMapping, @PostMapping, etc.) are convenient shortcuts.**

```
@RequestMapping("/api")
@RestController
class ProductApi {
 @GetMapping(value="/products", produces="application/json")
 List<Product> all(){ /* ... */ }

 @PostMapping(value="/products", consumes="application/json")
 Product create(@RequestBody Product p){ /* ... */ }
}
```

Note: Method-level mappings refine class-level mapping.

@PathVariable

- **Binds URI template variables to method parameters.**
- **Supports type conversion; name inferred if parameters have names.**

```
@GetMapping("/users/{id}")
public UserDto find(@PathVariable("id") Long userId){ /* ... */ }
```

@RequestParam

- Binds query/form parameters to handler method parameters.
- Use defaultValue and required=false for optional params.

```
@GetMapping("/search")
public List<UserDto> search(@RequestParam(defaultValue="10") int limit){ /* ... */ }
```

@RequestBody

- Binds HTTP request body to a parameter using message converters (JSON/XML).
- Combine with @Valid for validation.

```
@PostMapping("/orders")
public Order create(@Valid @RequestBody CreateOrderRequest req){ /* ... */ }
```

@ResponseStatus

- Sets the HTTP status code for a handler or on a custom exception.
- Useful for simple success/error semantics.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
class UserNotFound extends RuntimeException { }
```

@CrossOrigin

- Enables CORS on controllers/handlers; configure origins, methods, headers.
- Handy for SPAs/mobile apps consuming APIs.

```
@CrossOrigin(origins="https://app.example.com")
@GetMapping("/public-data")
public Data dto(){ /* ... */ }
```

@ExceptionHandler + @ControllerAdvice

- **Map exceptions to custom responses and status codes.**
- **@ControllerAdvice applies handlers globally across controllers.**

```
@ControllerAdvice
class GlobalErrors {
 @ExceptionHandler(UserNotFound.class)
 @ResponseStatus(HttpStatus.NOT_FOUND)
 ErrorDto handle(UserNotFound ex){ return new ErrorDto(ex.getMessage()); }
}
```

@Transactional

- **Defines transaction boundaries; control propagation/isolation, readOnly, rollback rules.**
- **Place on service methods; proxies apply to public methods by default.**

```
@Transactional(readOnly=true)
public UserDto findUser(Long id){ /* ... */ }
```

Note: Self-invocation bypasses proxies; call through another bean or enable AspectJ mode.

@Profile

- Activates beans only for specified profiles (e.g., dev, test, prod).
- Set active profiles via 'spring.profiles.active' property or env.

```
@Profile("dev")
@Bean DataSource h2(){ return new EmbeddedDatabaseBuilder().build(); }
```

@ConditionalOnProperty

- **Conditionally creates beans based on property presence/value.**
- **Useful for feature flags and optional integrations.**

```
@ConditionalOnProperty(prefix="feature.x", name="enabled", havingValue="true")
@Bean XService x(){ return new XService(); }
```

@Lazy

- **Defers bean initialization until first use.**
- **Can help with circular dependencies (use carefully).**

```
@Lazy
@Component
class HeavyClient { /* expensive init */ }
```

@Scope

- Defines bean scope: singleton (default), prototype, request, session, application, websocket.
- Lifecycle callbacks differ per scope.

```
@Scope("prototype")
@Bean Report report(){ return new Report(); }
```

@Valid (+ Bean Validation)

- Triggers JSR-380 validation on arguments/fields.
- Use constraints like @NotBlank, @Email, @Size; handle errors globally.

```
class SignupDto {
  @NotBlank String name;
  @Email String email;
  @Size(min=8) String password;
}

@PostMapping("/signup")
public void signup(@Valid @RequestBody SignupDto dto){ /* ... */ }
```

@Cacheable / @CachePut / @CacheEvict

- **@Cacheable caches method results by key (skips method if hit).**
- **@CachePut updates cache every invocation; @CacheEvict removes entries.**

```
@Cacheable(cacheNames="users", key="#id")
public User findUser(Long id){ /* DB */ }

@CacheEvict(cacheNames="users", key="#id")
public void deleteUser(Long id){ /* ... */ }
```

Note: Ensure proper equals/hashCode on key types; beware of caching nulls.

Actuator @Endpoint / @ReadOperation / @WriteOperation

- **Define custom Actuator endpoints for ops/monitoring.**
- **@ReadOperation maps to GET; @WriteOperation to POST/PUT.**

```
@Endpoint(id="features")
class FeaturesEndpoint {
 @ReadOperation
 public Map<String, Boolean> features(){ return Map.of("beta", true); }
}
```

Note: Secure endpoints; expose selectively via management.endpoints.web.exposure.include.