Annotation Reference

Spring Boot Developer Cheat Sheet v2.0 - Complete Annotation Reference

Table of Contents

---

1. Core Spring Annotations

@Component

Purpose: Marks a class as a Spring-managed component/bean.

Usage: Generic stereotype for any Spring-managed component.

Example:

```
@Component
public class EmailService {
    public void sendEmail(String to, String message) {
        // Implementation
    }
}
```

Details:

- Auto-detected through classpath scanning
- Can specify bean name: @Component("customName")
- Parent annotation for @Service, @Repository, @Controller

@Service

Purpose: Specialization of @Component for service layer classes.

Usage: Business logic layer components.

Example:

```
@Service
public class UserService {
    @Autowired
    private UserRepository repository;

    public User findById(Long id) {
        return repository.findById(id).orElseThrow();
    }
}
```

Details:

- Semantically indicates business service facade
- No additional behavior over @Component
- Makes code more readable and maintainable

@Repository

Purpose: Specialization of @Component for data access layer.

Usage: DAO/Repository classes that interact with database.

Example:

```
@Repository
public class UserRepository {
    @PersistenceContext
    private EntityManager entityManager;

    public User save(User user) {
        return entityManager.merge(user);
    }
}
```

Details:

- Enables automatic exception translation
- Converts database exceptions to Spring's DataAccessException
- Works with Spring's persistence exception translation

@Controller

Purpose: Marks class as Spring MVC controller.

Usage: Web layer components handling HTTP requests.

Example:

```
@Controller
public class ViewController {
    @GetMapping("/home")
    public String home(Model model) {
        model.addAttribute("message", "Welcome!");
        return "home"; // Returns view name
    }
}
```

Details:

- Handles HTTP requests and returns view names
- Combined with @ResponseBody becomes @RestController
- Supports model attributes and view resolution

@Autowired

Purpose: Automatic dependency injection.

Usage: Constructor, setter, or field injection.

Example:

```
@Service
public class OrderService {
    // Field injection (not recommended)
    @Autowired
    private PaymentService paymentService;

    // Constructor injection (recommended)
    private final UserService userService;

    @Autowired
    public OrderService(UserService userService) {
        this.userService = userService;
    }

    // Setter injection
    private NotificationService notificationService;

    @Autowired
    public void setNotificationService(NotificationService service) {
        this.notificationService = service;
    }
}
```

Details:

- Required by default (throws exception if bean not found)

- Can be made optional: @Autowired(required = false)
- Constructor injection doesn't need @Autowired (Spring 4.3+)

@Qualifier

Purpose: Specifies which bean to inject when multiple candidates exist.

Usage: Used with @Autowired to resolve ambiguity.

Example:

```
@Service
public class NotificationService {
    @Autowired
    @Qualifier("emailSender")
    private MessageSender emailSender;

    @Autowired
    @Qualifier("smsSender")
    private MessageSender smsSender;
}
```

Details:

- Bean name is used as default qualifier
- Custom qualifiers can be created
- Can be used on method parameters

@Primary

Purpose: Indicates preferred bean when multiple candidates exist.

Usage: Marks default bean for injection.

Example:

```
@Component
@Primary
public class PrimaryDataSource implements DataSource {
    // This will be injected by default
}

@Component
public class SecondaryDataSource implements DataSource {
    // This needs @Qualifier to be injected
}
```

Details:

- Reduces need for @Qualifier annotations
- Only one @Primary bean per type
- Overridden by explicit @Qualifier

@Bean

Purpose: Declares a method as bean producer.

Usage: Inside @Configuration classes to define beans.

Example:

```
@Configuration
public class AppConfig {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @Bean(name = "customExecutor")
    @Primary
    public Executor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.initialize();
        return executor;
    }
}
```

Details:

- Method name becomes bean name by default
- Supports init/destroy methods: @Bean(initMethod = "init", destroyMethod = "cleanup")
- Can specify multiple names: @Bean({"name1", "name2"})

@Scope

Purpose: Defines bean lifecycle scope.

Usage: Controls bean instantiation strategy.

Example:

```
@Component
@Scope("prototype")
public class PrototypeBean {
    // New instance created for each injection
}

@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class SessionBean {
    // One instance per HTTP session
}
```

Details:

- Available scopes: singleton (default), prototype, request, session, application
- Custom scopes can be created
- Proxy mode needed for injecting shorter-lived scopes into longer-lived beans

@PostConstruct & @PreDestroy

Purpose: Lifecycle callback methods.

Usage: Initialization and cleanup logic.

Example:

```
@Component
public class CacheManager {
    private Map<String, Object> cache;

    @PostConstruct
    public void init() {
        cache = new HashMap<>();
        loadInitialData();
    }

    @PreDestroy
    public void cleanup() {
        cache.clear();
        releaseResources();
    }
}
```

Details:

- @PostConstruct called after dependency injection
- @PreDestroy called before bean destruction
- Not called for prototype scoped beans

@Lazy

Purpose: Delays bean initialization until first use.

Usage: Optimize startup time or break circular dependencies.

Example:

```
@Service
@Lazy
public class ExpensiveService {
    // Initialized only when first accessed
}

@Service
public class StartupService {
    @Autowired
    @Lazy
    private ExpensiveService expensiveService;
}
```

Details:

- Can be used on @Bean methods
- Helps with circular dependency resolution
- Improves application startup time

---

2. Spring Boot Annotations

@SpringBootApplication

Purpose: Main annotation combining multiple annotations.

Usage: Main class of Spring Boot application.

Example:

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

Details:

- Combines @Configuration, @EnableAutoConfiguration, @ComponentScan
- Can exclude specific auto-configurations
- Can specify component scan base packages

@EnableAutoConfiguration

Purpose: Enables Spring Boot's auto-configuration.

Usage: Automatically configures beans based on classpath.

Example:

```
@Configuration
@EnableAutoConfiguration(
    exclude = {SecurityAutoConfiguration.class}
)
public class CustomConfig {
    // Custom configuration
}
```

Details:

- Attempts to configure beans automatically
- Based on jars in classpath
- Can be fine-tuned with properties

@ConfigurationProperties

Purpose: Binds external properties to POJO.

Usage: Type-safe configuration properties.

Example:

```
@Component
@ConfigurationProperties(prefix = "app.mail")
@Validated
public class MailProperties {
    @NotBlank
    private String host;

    @Min(1)
    @Max(65535)
    private int port = 25;

    private Security security = new Security();

    // Getters and setters

    public static class Security {
        private boolean enabled;
        private String protocol = "TLS";
        // Getters and setters
    }
}
```

Details:

- Supports nested properties and collections
- Works with validation annotations
- Requires @EnableConfigurationProperties or @Component

@ConditionalOnProperty

Purpose: Conditional bean creation based on properties.

Usage: Feature toggles and environment-specific beans.

Example:

```
@Configuration
@ConditionalOnProperty(
    prefix = "app.feature",
    name = "cache.enabled",
    havingValue = "true",
    matchIfMissing = false
)
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        return new ConcurrentMapCacheManager();
    }
}
```

Details:

- Can check property existence or specific values
- Supports matching if property is missing
- Useful for feature flags

@ConditionalOnClass

Purpose: Creates bean only if class is present.

Usage: Library-specific auto-configuration.

Example:

```
@Configuration
@ConditionalOnClass(RedisTemplate.class)
public class RedisConfig {
    @Bean
    @ConditionalOnMissingBean
    public RedisTemplate<String, Object> redisTemplate() {
        // Configure Redis template
    }
}
```

Details:

- Checks classpath for specific classes

- Used heavily in auto-configuration
- Prevents ClassNotFoundException

@ConditionalOnMissingBean

Purpose: Creates bean only if not already defined.

Usage: Default bean configurations.

Example:

```
@Configuration
public class DefaultConfig {
    @Bean
    @ConditionalOnMissingBean(MessageService.class)
    public MessageService defaultMessageService() {
        return new EmailMessageService();
    }
}
```

Details:

- Allows user-defined beans to override defaults
- Can specify bean type or name
- Common in starter configurations

---

3. Spring Security Annotations

@EnableWebSecurity

Purpose: Enables Spring Security configuration.

Usage: Main security configuration class.

Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) {
        return http
            .authorizeHttpRequests(auth → auth
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .build();
    }
}
```

Details:

- Imports Spring Security configuration
- Enables security filter chain
- Must be used with @Configuration

@EnableMethodSecurity

Purpose: Enables method-level security annotations.

Usage: Allows @PreAuthorize, @PostAuthorize, etc.

Example:

```
@Configuration
@EnableMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true
)
public class MethodSecurityConfig {
    // Configuration
}
```

Details:

- Replaces deprecated @EnableGlobalMethodSecurity
- Enables different annotation styles

- Can configure custom permission evaluator

@PreAuthorize

Purpose: Checks authorization before method execution.

Usage: Method-level access control with SpEL.

Example:

```
@Service
public class DocumentService {
   @PreAuthorize("hasRole('ADMIN')")
   public void deleteAll() {
      // Admin only
   }

   @PreAuthorize("hasRole('USER') and #document.owner == authenticatio
n.name")
   public void updateDocument(Document document) {
      // User must own the document
   }

   @PreAuthorize("@securityService.hasAccess(#id)")
   public Document getDocument(Long id) {
      // Custom security check
   }
}
```

Details:

- Supports complex SpEL expressions
- Can reference method parameters
- Can call other beans for custom logic

@PostAuthorize

Purpose: Checks authorization after method execution.

Usage: Verify access to return value.

Example:

```
@Service
public class UserService {
    @PostAuthorize("returnObject.username == authentication.name")
    public User getCurrentUser() {
        // Ensures users can only access their own data
    }

    @PostAuthorize("hasRole('ADMIN') or returnObject.public")
    public Document getDocument(Long id) {
        // Admin can see all, others only public documents
    }
}
```

Details:

- Has access to return value
- Executes after method completion
- Can prevent returning sensitive data

@Secured

Purpose: Simple role-based security.

Usage: Basic role checking without SpEL.

Example:

```
@RestController
public class AdminController {
    @Secured("ROLE_ADMIN")
    @GetMapping("/admin/users")
    public List<User> getAllUsers() {
        // Admin only endpoint
    }

    @Secured({"ROLE_USER", "ROLE_ADMIN"})
    @GetMapping("/profile")
    public Profile getProfile() {
        // User or Admin
    }
}
```

Details:

- Less flexible than @PreAuthorize
- No SpEL support
- Requires exact role names

@RolesAllowed

Purpose: JSR-250 standard security annotation.

Usage: Java EE compatible role checking.

Example:

```
@Service
public class PaymentService {
    @RolesAllowed("ADMIN")
    public void refundAll() {
        // Implementation
    }

    @RolesAllowed({"USER", "ADMIN"})
    public Payment getPayment(Long id) {
        // Implementation
    }
}
```

Details:

- Standard Java annotation
- Requires jsr250Enabled = true
- Similar to @Secured

@AuthenticationPrincipal

Purpose: Injects authenticated user into method.

Usage: Access current user in controllers.

Example:

```
@RestController
public class UserController {
    @GetMapping("/me")
    public UserDto getCurrentUser(@AuthenticationPrincipal UserDetails us
er) {
        return userService.findByUsername(user.getUsername());
    }

    @GetMapping("/my-profile")
    public Profile getMyProfile(@AuthenticationPrincipal(expression = "id") L
ong userId) {
        return profileService.findByUserId(userId);
    }
}
```

Details:

- Replaces SecurityContextHolder access
- Can use SpEL to extract specific fields
- Null if not authenticated

---

4. Spring Web/REST Annotations

@RestController

Purpose: Combines @Controller and @ResponseBody.

Usage: RESTful web services returning data.

Example:

```
@RestController
@RequestMapping("/api/v1/products")
public class ProductController {
    @GetMapping
    public List<Product> getAllProducts() {
        return productService.findAll(); // Returns JSON
    }
}
```

Details:

- Every method returns data, not view names
- Automatic JSON/XML serialization
- No need for @ResponseBody on each method

@RequestMapping

Purpose: Maps HTTP requests to handler methods.

Usage: Class or method level URL mapping.

Example:

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @RequestMapping(
        value = "/{id}",
        method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE
    )
    public User getUser(@PathVariable Long id) {
        return userService.findById(id);
    }

    @RequestMapping(
        method = RequestMethod.POST,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public User createUser(@RequestBody User user) {
        return userService.create(user);
    }
}
```

Details:

- Supports all HTTP methods
- Can specify content types
- Supports multiple URLs

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

Purpose: Specialized request mappings for specific HTTP methods.

Usage: Shorthand for @RequestMapping with method.

Example:

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {
    @GetMapping("/{id}")
    public Order getOrder(@PathVariable Long id) {
        return orderService.findById(id);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Order createOrder(@Valid @RequestBody OrderRequest request)
{
        return orderService.create(request);
    }

    @PutMapping("/{id}")
    public Order updateOrder(@PathVariable Long id, @Valid @RequestBody
OrderRequest request) {
        return orderService.update(id, request);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteOrder(@PathVariable Long id) {
        orderService.delete(id);
    }

    @PatchMapping("/{id}/status")
    public Order updateStatus(@PathVariable Long id, @RequestParam Orde
rStatus status) {
        return orderService.updateStatus(id, status);
    }
}
```

Details:

- More readable than @RequestMapping
- Same attributes available
- Clearly indicates HTTP method

@PathVariable

Purpose: Binds URI template variables to method parameters.

Usage: Extract values from URL path.

Example:

```
@GetMapping("/users/{userId}/orders/{orderId}")
public Order getUserOrder(
    @PathVariable Long userId,
    @PathVariable("orderId") Long id,  // Different parameter name
    @PathVariable Map<String, String> allPathVars  // All variables
) {
    // userId and orderId are extracted from URL
    return orderService.findByUserAndId(userId, id);
}
```

```
@GetMapping("/files/{*path}")  // Captures remaining path
public Resource getFile(@PathVariable String path) {
    return fileService.load(path);
}
```

Details:

- Required by default
- Can be made optional in Spring 4.3+
- Supports regex patterns

@RequestParam

Purpose: Binds query parameters to method arguments.

Usage: Extract query string parameters.

Example:

```
@GetMapping("/search")
public Page<Product> searchProducts(
    @RequestParam(required = false) String query,
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "20") int size,
    @RequestParam(name = "sort_by", defaultValue = "name") String sortB
y,
    @RequestParam Map<String, String> allParams  // All parameters
) {
    return productService.search(query, page, size, sortBy);
}

@GetMapping("/filter")
public List<Product> filterProducts(
    @RequestParam List<String> categories,  // Multiple values
    @RequestParam Optional<BigDecimal> maxPrice  // Optional
) {
    return productService.filter(categories, maxPrice.orElse(null));
}
```

Details:

- Can have default values
- Supports collections for multiple values
- Automatic type conversion

@RequestBody

Purpose: Binds HTTP request body to Java object.

Usage: Deserialize JSON/XML to objects.

Example:

```
@PostMapping("/users")
public User createUser(@Valid @RequestBody UserRequest request) {
    // JSON is automatically converted to UserRequest object
    return userService.create(request);
}

@PutMapping("/bulk")
public List<User> updateBulk(@RequestBody List<UserRequest> requests)
{
    // Handles array of objects
    return userService.updateAll(requests);
}

@PostMapping("/raw")
public void processRaw(@RequestBody String rawData) {
    // Raw request body as string
    processService.handleRaw(rawData);
}
```

Details:

- Uses HttpMessageConverters

- Works with @Valid for validation
- One @RequestBody per method

@ResponseBody

Purpose: Writes method return value to response body.

Usage: Return data instead of view name.

Example:

```
@Controller  // Not @RestController
public class DataController {
    @GetMapping("/data")
    @ResponseBody
    public Map<String, Object> getData() {
        return Map.of("status", "success", "data", Arrays.asList(1, 2, 3));
    }

    @GetMapping("/page")
    public String getPage(Model model) {
        // This returns a view name, not data
        return "homepage";
    }
}
```

Details:

- Implicit in @RestController
- Uses content negotiation
- Bypasses view resolution

@ResponseStatus

Purpose: Sets HTTP response status code.

Usage: Declare response status for methods or exceptions.

Example:

```
@PostMapping("/items")
@ResponseStatus(HttpStatus.CREATED)  // Returns 201
public Item createItem(@RequestBody Item item) {
    return itemService.create(item);
}

@DeleteMapping("/items/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)  // Returns 204
public void deleteItem(@PathVariable Long id) {
    itemService.delete(id);
}

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    // Exception always returns 404
}
```

Details:

- Can include reason phrase
- Works on exception classes
- Overrides default status codes

@RequestHeader

Purpose: Binds HTTP header to method parameter.

Usage: Access request headers.

Example:

```
@GetMapping("/data")
public Data getData(
    @RequestHeader("Authorization") String auth,
    @RequestHeader(value = "X-Request-ID", required = false) String reque
stId,
    @RequestHeader HttpHeaders allHeaders,
    @RequestHeader Map<String, String> headerMap
) {
    // Access individual or all headers
    return dataService.getData(auth);
}
```

Details:

- Can specify header name
- Supports required/optional
- Can inject all headers

@CookieValue

Purpose: Binds cookie value to method parameter.

Usage: Extract cookie values.

Example:

```
@GetMapping("/preferences")
public Preferences getPreferences(
    @CookieValue(value = "sessionId", defaultValue = "") String sessionId,
    @CookieValue(required = false) String theme
) {
    return preferenceService.get(sessionId, theme);
}
```

Details:

- Can have default values
- Optional cookies supported
- Automatic type conversion

@ModelAttribute

Purpose: Binds request parameters to object or adds to model.

Usage: Form data binding and model population.

Example:

```
@Controller
public class FormController {
    @ModelAttribute("categories")
    public List<String> populateCategories() {
        // This runs before every handler method
        return categoryService.getAllCategories();
    }

    @PostMapping("/submit")
    public String submitForm(@ModelAttribute("form") @Valid FormData for
mData,
                           BindingResult result) {
        if (result.hasErrors()) {
            return "form";  // Return to form view
        }
        return "success";
    }
}
```

Details:

- Runs before handler methods
- Useful for form backing objects
- Adds attributes to model

@SessionAttributes

Purpose: Stores model attributes in HTTP session.

Usage: Maintain state across requests.

Example:

```
@Controller
@SessionAttributes({"user", "preferences"})
public class WizardController {
    @GetMapping("/step1")
    public String step1(Model model) {
        model.addAttribute("user", new User());
        return "step1";
    }

    @PostMapping("/step2")
    public String step2(@ModelAttribute("user") User user) {
        // User object maintained in session
        return "step2";
    }

    @GetMapping("/complete")
    public String complete(SessionStatus status) {
        status.setComplete();  // Clear session attributes
        return "complete";
    }
}
```

Details:

- Survives across requests
- Cleared with SessionStatus
- Use carefully to avoid memory issues

@CrossOrigin

Purpose: Enables Cross-Origin Resource Sharing (CORS).

Usage: Allow cross-domain requests.

Example:

```
@RestController
@CrossOrigin(origins = "http://localhost:3000")
public class ApiController {
    // All endpoints allow CORS from localhost:3000
}

@CrossOrigin(
    origins = {"http://app.example.com", "https://app.example.com"},
    methods = {RequestMethod.GET, RequestMethod.POST},
    allowedHeaders = {"Authorization", "Content-Type"},
    exposedHeaders = {"X-Total-Count"},
    allowCredentials = "true",
    maxAge = 3600
)
@GetMapping("/data")
public List<Data> getData() {
    return dataService.findAll();
}
```

Details:

- Can be class or method level
- Global CORS configuration available
- Important for frontend integration

---

5. Spring Data/JPA Annotations

@Entity

Purpose: Marks class as JPA entity.

Usage: Domain objects mapped to database tables.

Example:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true, length = 100)
    private String email;

    @Column(name = "full_name")
    private String fullName;

    @Enumerated(EnumType.STRING)
    private UserStatus status;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdAt;

    // For Java 8 time API
    private LocalDateTime updatedAt;
}
```

Details:

- Must have @Id field
- Default table name is class name
- Requires no-arg constructor

@Table

Purpose: Specifies table details for entity.

Usage: Customize table mapping.

Example:

```
@Entity
@Table(
    name = "user_accounts",
    schema = "public",
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"email"}),
        @UniqueConstraint(columnNames = {"username", "tenant_id"})
    },
    indexes = {
        @Index(name = "idx_email", columnList = "email"),
        @Index(name = "idx_created", columnList = "created_at DESC")
    }
)
```

```
public class UserAccount {
    // Entity fields
}
```

Details:

- Can specify schema/catalog
- Define indexes and constraints
- Useful for database generation

@Id

Purpose: Marks primary key field.

Usage: Required for every entity.

Example:

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Composite key example
    @EmbeddedId
    private OrderItemId id;

    // ID class example
    @Id
    private Long orderId;
    @Id
    private Long productId;
}
```

Details:

- Can be primitive or wrapper type
- Supports composite keys

- Must be unique

@GeneratedValue

Purpose: Configures primary key generation.

Usage: Auto-generate ID values.

Example:

```
// AUTO - JPA picks strategy
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;

// IDENTITY - Database identity column
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

// SEQUENCE - Database sequence
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "us
er_seq")
@SequenceGenerator(name = "user_seq", sequenceName = "user_sequen
ce", allocationSize = 1)
private Long id;

// TABLE - Separate table for IDs
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "user_ge
n")
@TableGenerator(name = "user_gen", table = "id_generator", pkColumnVal
ue = "user_id")
private Long id;

// UUID example
@Id
@GeneratedValue(generator = "uuid2")
```

```
@GenericGenerator(name = "uuid2", strategy = "uuid2")
private String id;
```

Details:

- Strategy depends on database
- IDENTITY doesn't support batch inserts
- SEQUENCE most efficient for batch

@Column

Purpose: Customizes column mapping.

Usage: Configure column properties.

Example:

```
@Entity
public class Employee {
    @Column(
        name = "emp_name",
        nullable = false,
        length = 100,
        unique = true
    )
    private String name;

    @Column(
        precision = 10,
        scale = 2,
        columnDefinition = "DECIMAL(10,2) DEFAULT 0.00"
    )
    private BigDecimal salary;

    @Column(
        insertable = false,
        updatable = false,
        columnDefinition = "TIMESTAMP DEFAULT CURRENT_TIMESTAMP"
    )
    private LocalDateTime createdAt;
}
```

Details:

- Controls DDL generation
- Can specify SQL types
- Insert/update control

@OneToMany, @ManyToOne, @OneToOne, @ManyToMany

Purpose: Define entity relationships.

Usage: Map associations between entities.

Example:

```
@Entity
public class Author {
    @Id
    private Long id;

    // One author has many books
    @OneToMany(
        mappedBy = "author",
        cascade = CascadeType.ALL,
        orphanRemoval = true,
        fetch = FetchType.LAZY
    )
    private List<Book> books = new ArrayList<>();

    // Bidirectional OneToOne
    @OneToOne(
        mappedBy = "author",
        cascade = CascadeType.ALL,
        fetch = FetchType.LAZY,
        optional = false
    )
    @PrimaryKeyJoinColumn
    private AuthorDetails details;
}

@Entity
public class Book {
    @Id
    private Long id;

    // Many books belong to one author
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "author_id", nullable = false)
    private Author author;

    // Many-to-many with join table
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERG
E})
    @JoinTable(
        name = "book_category",
        joinColumns = @JoinColumn(name = "book_id"),
        inverseJoinColumns = @JoinColumn(name = "category_id")
    )
    private Set<Category> categories = new HashSet<>();
}
```

Details:

- Default fetch: EAGER for ToOne, LAZY for ToMany
- mappedBy indicates non-owning side
- Cascade operations carefully

@JoinColumn

Purpose: Specifies foreign key column.

Usage: Configure join column properties.

Example:

```
@Entity
public class Order {
    @ManyToOne
    @JoinColumn(
        name = "customer_id",
        referencedColumnName = "id",
        nullable = false,
        foreignKey = @ForeignKey(name = "fk_order_customer")
    )
    private Customer customer;

    @OneToOne
    @JoinColumns({
        @JoinColumn(name = "shipping_address_id", referencedColumnName
= "id"),
        @JoinColumn(name = "shipping_country", referencedColumnName =
"country")
    })
    private Address shippingAddress;
}
```

Details:

- Specifies foreign key details
- Can reference non-primary keys
- Supports composite keys

@Transactional

Purpose: Manages database transactions.

Usage: Declarative transaction management.

Example:

```
@Service
@Transactional(readOnly = true)  // Class level default
public class UserService {
    @Transactional  // Override for write operation
    public User createUser(UserDto dto) {
        User user = new User(dto);
        return userRepository.save(user);
    }

    @Transactional(
        propagation = Propagation.REQUIRES_NEW,
        isolation = Isolation.READ_COMMITTED,
        timeout = 30,
        rollbackFor = {BusinessException.class},
        noRollbackFor = {ValidationException.class}
    )
    public void complexOperation() {
        // Runs in new transaction
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void requiresExistingTransaction() {
        // Must be called within transaction
    }
}
```

Details:

- Default rollback on RuntimeException
- Proxy-based - won't work on private methods
- Various propagation levels available

@Query

Purpose: Defines custom queries.

Usage: JPQL or native SQL queries.

Example:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // JPQL query
    @Query("SELECT u FROM User u WHERE u.email = ?1")
    Optional<User> findByEmail(String email);

    // Named parameters
    @Query("SELECT u FROM User u WHERE u.status = :status AND u.creat
edAt > :date")
    List<User> findActiveUsersSince(@Param("status") Status status, @Par
am("date") LocalDate date);

    // Native SQL
    @Query(
        value = "SELECT * FROM users WHERE YEAR(created_at) = ?1",
        nativeQuery = true
    )
    List<User> findByYear(int year);

    // DTO projection
    @Query("SELECT new com.example.dto.UserSummary(u.id, u.name, u.e
mail) FROM User u")
    List<UserSummary> findAllSummaries();

    // Pageable support
    @Query(
        value = "SELECT u FROM User u WHERE u.active = true",
        countQuery = "SELECT COUNT(u) FROM User u WHERE u.active = tru
e"
    )
    Page<User> findActiveUsers(Pageable pageable);
}
```

Details:

- Supports SpEL expressions
- Can return various types

- Validated at startup

@Modifying

Purpose: Indicates query modifies data.

Usage: UPDATE/DELETE queries.

Example:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
   @Modifying
   @Query("UPDATE User u SET u.active = false WHERE u.lastLogin < :dat
e")
   int deactivateInactiveUsers(@Param("date") LocalDate date);

   @Modifying(clearAutomatically = true)  // Clear persistence context
   @Query("UPDATE User u SET u.credits = u.credits + :amount WHERE u.i
d = :id")
   void addCredits(@Param("id") Long id, @Param("amount") int amount);

   @Modifying
   @Query(value = "DELETE FROM user_sessions WHERE user_id = ?1", nat
iveQuery = true)
   void deleteUserSessions(Long userId);
}
```

Details:

- Required for UPDATE/DELETE
- Returns affected row count
- Can clear persistence context

@EntityGraph

Purpose: Defines fetch plan for queries.

Usage: Solve N+1 query problems.

Example:

```
@Entity
@NamedEntityGraph(
    name = "User.withOrdersAndItems",
    attributeNodes = {
        @NamedAttributeNode("profile"),
        @NamedAttributeNode(value = "orders", subgraph = "order-items")
    },
    subgraphs = {
        @NamedSubgraph(
            name = "order-items",
            attributeNodes = @NamedAttributeNode("items")
        )
    }
)
public class User {
    // Entity definition
}

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Using named entity graph
    @EntityGraph("User.withOrdersAndItems")
    Optional<User> findWithOrdersById(Long id);

    // Ad-hoc entity graph
    @EntityGraph(attributePaths = {"profile", "orders"})
    List<User> findByActiveTrue();

    // Override fetch type
    @EntityGraph(type = EntityGraph.EntityGraphType.LOAD)
    List<User> findAll();
}
```

Details:

- Prevents N+1 queries

- FETCH type overwrites mappings
- LOAD type uses mappings as default

---

6. Spring Cloud Annotations

@EnableEurekaClient / @EnableDiscoveryClient

Purpose: Registers service with discovery server.

Usage: Microservice registration.

Example:

```
@SpringBootApplication
@EnableEurekaClient  // Specific to Eureka
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}

// Configuration
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
    instance-id: ${spring.application.name}:${random.value}
```

Details:

- Auto-registers on startup
- Sends heartbeats
- @EnableDiscoveryClient is generic

@FeignClient

Purpose: Declarative REST client.

Usage: Service-to-service communication.

Example:

```
@FeignClient(
    name = "user-service",
    url = "${user.service.url}",  // Optional, uses discovery by default
    configuration = FeignConfig.class,
    fallback = UserClientFallback.class
)
public interface UserClient {
    @GetMapping("/api/users/{id}")
    User getUser(@PathVariable("id") Long id);

    @PostMapping("/api/users")
    User createUser(@RequestBody UserDto dto);

    @GetMapping("/api/users")
    Page<User> getUsers(@RequestParam("page") int page,
                       @RequestParam("size") int size);
}

@Component
public class UserClientFallback implements UserClient {
    @Override
    public User getUser(Long id) {
        return new User("Fallback User");
    }
    // Other fallback methods
}
```

Details:

- Built-in load balancing
- Integrates with Hystrix/Resilience4j
- Supports custom configuration

@LoadBalanced

Purpose: Enables client-side load balancing.

Usage: With RestTemplate or WebClient.

Example:

```
@Configuration
public class RestConfig {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @Bean
    @LoadBalanced
    public WebClient.Builder webClientBuilder() {
        return WebClient.builder();
    }
}

@Service
public class OrderService {
    @Autowired
    private RestTemplate restTemplate;

    public User getUser(Long userId) {
        // Service name instead of URL
        return restTemplate.getForObject(
            "http://user-service/api/users/{id}",
            User.class,
            userId
        );
    }
}
```

Details:

- Uses Ribbon/Spring Cloud LoadBalancer
- Service discovery integration
- Round-robin by default

@EnableConfigServer

Purpose: Creates configuration server.

Usage: Centralized configuration management.

Example:

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

// application.yml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/config-repo
          search-paths: '{application}'
          default-label: main
        encrypt:
          enabled: true
```

Details:

- Serves configuration from Git/SVN/filesystem
- Supports encryption/decryption
- Environment-specific configs

@RefreshScope

Purpose: Enables configuration refresh without restart.

Usage: Dynamic configuration updates.

Example:

```
@RestController
@RefreshScope
public class ConfigController {
    @Value("${app.message}")
    private String message;

    @Value("${app.feature.enabled:false}")
    private boolean featureEnabled;

    @GetMapping("/message")
    public String getMessage() {
        return message;  // Updated on refresh
    }
}

// Refresh via actuator
// POST /actuator/refresh
```

Details:

- Creates proxy for bean
- Refreshed on /refresh endpoint
- Works with @ConfigurationProperties

@EnableCircuitBreaker

Purpose: Enables circuit breaker pattern.

Usage: Fault tolerance in microservices.

Example:

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {
    // Main method
}

@Service
public class UserService {
    @HystrixCommand(
        fallbackMethod = "getDefaultUser",
        commandProperties = {
            @HystrixProperty(name = "execution.isolation.thread.timeoutInMillis
econds", value = "3000"),
            @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",
value = "10"),
            @HystrixProperty(name = "circuitBreaker.errorThresholdPercentag
e", value = "50")
        }
    )
    public User getUser(Long id) {
        // Call external service
    }

    public User getDefaultUser(Long id) {
        return new User("Default User");
    }
}
```

Details:

- Hystrix deprecated, use Resilience4j
- Prevents cascading failures
- Monitors and breaks circuit

@CircuitBreaker (Resilience4j)

Purpose: Modern circuit breaker implementation.

Usage: Resilience pattern for microservices.

Example:

```
@Service
public class PaymentService {
   @CircuitBreaker(name = "payment-service", fallbackMethod = "fallback
Payment")
   @Retry(name = "payment-service")
   @RateLimiter(name = "payment-service")
   public PaymentResult processPayment(PaymentRequest request) {
      // External payment gateway call
   }

   public PaymentResult fallbackPayment(PaymentRequest request, Excepti
on ex) {
      log.error("Payment failed, using fallback", ex);
      return PaymentResult.failed("Service temporarily unavailable");
   }
}

// Configuration
resilience4j:
  circuitbreaker:
    instances:
      payment-service:
        sliding-window-size: 10
        failure-rate-threshold: 50
        wait-duration-in-open-state: 30s
        permitted-number-of-calls-in-half-open-state: 3
  retry:
    instances:
      payment-service:
        max-attempts: 3
        wait-duration: 1s
```

Details:

- Replaces Hystrix

- Multiple resilience patterns
- Metrics integration

---

7. Testing Annotations

@SpringBootTest

Purpose: Integration testing with full context.

Usage: Load complete application context.

Example:

```
@SpringBootTest(
    classes = {TestConfig.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    properties = {
        "spring.datasource.url=jdbc:h2:mem:testdb",
        "app.feature.enabled=true"
    }
)
@ActiveProfiles("test")
@TestPropertySource(locations = "classpath:test.properties")
class ApplicationIntegrationTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void contextLoads() {
        assertThat(restTemplate).isNotNull();
    }
}
```

Details:

- Heavy but comprehensive
- Various web environments
- Can override properties

@WebMvcTest

Purpose: Test Spring MVC controllers.

Usage: Focused web layer testing.

Example:

```
@WebMvcTest(UserController.class)
@Import(SecurityConfig.class)  // If needed
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    @WithMockUser(roles = "ADMIN")
    void getUserById_ReturnsUser() throws Exception {
        // Given
        User user = new User(1L, "John");
        when(userService.findById(1L)).thenReturn(user);

        // When & Then
        mockMvc.perform(get("/api/users/1")
                .accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name").value("John"))
            .andDo(print());
    }
}
```

Details:

- Only loads web layer
- Auto-configures MockMvc
- Fast execution

@DataJpaTest

Purpose: Test JPA repositories.

Usage: Database layer testing.

Example:

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Repla
ce.NONE)
@Sql({"/schema.sql", "/test-data.sql"})
class UserRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository userRepository;

    @Test
    @Rollback(false)  // Keep data for debugging
    void findByEmail_ReturnsUser() {
        // Given
        User user = new User("test@example.com");
        entityManager.persistAndFlush(user);
        entityManager.clear();  // Clear cache

        // When
        Optional<User> found = userRepository.findByEmail("test@example.c
om");

        // Then
        assertThat(found).isPresent();
        assertThat(found.get().getEmail()).isEqualTo("test@example.com");
    }
}
```

Details:

- Uses embedded database by default
- Transactional with rollback
- Includes TestEntityManager

@MockBean

Purpose: Mock Spring beans in tests.

Usage: Replace beans with Mockito mocks.

Example:

```
@SpringBootTest
class OrderServiceIntegrationTest {

    @MockBean
    private PaymentGateway paymentGateway;

    @SpyBean  // Partial mock
    private NotificationService notificationService;

    @Autowired
    private OrderService orderService;

    @Test
    void createOrder_Success() {
        // Given
        when(paymentGateway.process(any())).thenReturn(PaymentResult.su
ccess());
        doNothing().when(notificationService).sendEmail(any());

        // When
        Order order = orderService.create(new OrderRequest());

        // Then
        assertThat(order).isNotNull();
        verify(paymentGateway).process(any());
        verify(notificationService).sendEmail(any());
    }
}
```

Details:

- Replaces existing beans
- Reset after each test
- Works with @SpyBean

@TestConfiguration

Purpose: Additional configuration for tests.

Usage: Test-specific beans.

Example:

```
@TestConfiguration
public class TestConfig {

    @Bean
    @Primary
    public Clock testClock() {
        return Clock.fixed(Instant.parse("2023-01-01T00:00:00Z"), ZoneOffse
t.UTC);
    }

    @Bean
    public RestTemplateBuilder restTemplateBuilder() {
        return new RestTemplateBuilder()
            .setConnectTimeout(Duration.ofSeconds(1))
            .setReadTimeout(Duration.ofSeconds(1));
    }
}

@SpringBootTest
@Import(TestConfig.class)
class TimeBasedTest {
    @Autowired
    private Clock clock;

    // Tests use fixed time
}
```

Details:

- Doesn't replace main config
- Can override specific beans
- Scanned automatically in tests

@TestPropertySource

Purpose: Override properties in tests.

Usage: Test-specific configuration.

Example:

```
@SpringBootTest
@TestPropertySource(
    locations = "classpath:test.properties",
    properties = {
        "app.feature.new-algorithm=true",
        "spring.cache.type=none",
        "logging.level.com.example=DEBUG"
    }
)
class FeatureTest {
    @Value("${app.feature.new-algorithm}")
    private boolean newAlgorithm;

    @Test
    void testNewAlgorithm() {
        assertThat(newAlgorithm).isTrue();
    }
}
```

Details:

- Properties override application.properties
- Inline properties highest precedence
- Useful for feature flags

@DirtiesContext

Purpose: Reset application context.

Usage: When test modifies context state.

Example:

```
@SpringBootTest
class CacheTest {

    @Test
    @DirtiesContext(methodMode = DirtiesContext.MethodMode.AFTER_ME
THOD)
    void testThatModifiesCache() {
        // Test that pollutes cache
    }

    @Test
    @DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_CLASS)
    void anotherTest() {
        // Clean cache after all tests in class
    }
}
```

Details:

- Expensive operation

- Use sparingly
- Various modes available

@AutoConfigureMockMvc

Purpose: Auto-configure MockMvc.

Usage: Web testing with full context.

Example:

```
@SpringBootTest
@AutoConfigureMockMvc(
    addFilters = true,  // Include security filters
    print = MockMvcPrint.SYSTEM_ERR,
    printOnlyOnFailure = false
)
class WebIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    @WithUserDetails("admin@example.com")
    void securedEndpoint_ReturnsData() throws Exception {
        mockMvc.perform(get("/api/admin/users"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$").isArray());
    }
}
```

Details:

- Full context with MockMvc
- Includes all filters
- Good for security testing

---

8. Validation Annotations

@Valid

Purpose: Triggers validation on object.

Usage: Method parameters and fields.

Example:

```
@RestController
public class UserController {
    @PostMapping("/users")
    public User createUser(@Valid @RequestBody UserDto dto) {
        // Validation triggered before method execution
        return userService.create(dto);
    }

    @PutMapping("/users/{id}")
    public User updateUser(
        @PathVariable Long id,
        @Valid @RequestBody UpdateUserDto dto,
        BindingResult bindingResult  // Contains validation errors
    ) {
        if (bindingResult.hasErrors()) {
            // Custom error handling
        }
        return userService.update(id, dto);
    }
}
```

Details:

- Triggers Bean Validation
- Works with @Validated
- Throws MethodArgumentNotValidException

@Validated

Purpose: Spring's validation annotation with groups.

Usage: Class level or method parameter.

Example:

```
@RestController
@Validated  // Enables validation for all methods
public class PaymentController {

    @PostMapping("/payments")
    public Payment createPayment(
        @Validated(CreateGroup.class) @RequestBody PaymentDto dto
    ) {
        return paymentService.create(dto);
    }

    @PutMapping("/payments/{id}")
    public Payment updatePayment(
        @PathVariable Long id,
        @Validated(UpdateGroup.class) @RequestBody PaymentDto dto
    ) {
        return paymentService.update(id, dto);
    }
}

public class PaymentDto {
    @NotNull(groups = {CreateGroup.class, UpdateGroup.class})
    private BigDecimal amount;

    @NotNull(groups = CreateGroup.class)
    private String accountNumber;

    @AssertTrue(groups = UpdateGroup.class)
    private boolean confirmed;
}
```

Details:

- Supports validation groups
- More features than @Valid

- Can be used on classes

@NotNull, @NotEmpty, @NotBlank

Purpose: Null and empty checks.

Usage: Field validation.

Example:

```
public class UserDto {
    @NotNull(message = "ID cannot be null")
    private Long id;

    @NotEmpty(message = "List must have at least one element")
    private List<String> roles;  // Not null and size > 0

    @NotBlank(message = "Name must not be blank")
    private String name;  // Not null, trimmed length > 0

    // @NotNull: value != null
    // @NotEmpty: value != null && value.length() > 0
    // @NotBlank: value != null && value.trim().length() > 0
}
```

Details:

- @NotBlank only for Strings
- @NotEmpty for Strings, Collections, Maps, Arrays
- Custom messages supported

@Size

Purpose: Validates size/length.

Usage: Strings, Collections, Arrays.

Example:

```
public class ProductDto {
    @Size(min = 3, max = 100, message = "Name must be between {min} an
d {max} characters")
    private String name;

    @Size(min = 1, max = 5)
    private List<String> categories;

    @Size(max = 1000)
    private String description;
}
```

Details:

- Inclusive boundaries
- Works with collections
- Null values pass validation

@Min, @Max

Purpose: Numeric range validation.

Usage: Numeric fields.

Example:

```
public class OrderDto {
    @Min(value = 1, message = "Quantity must be at least 1")
    private Integer quantity;

    @Max(value = 100, message = "Cannot order more than 100 items")
    private Integer maxQuantity;

    @DecimalMin(value = "0.01", inclusive = true)
    private BigDecimal price;

    @DecimalMax(value = "999999.99", inclusive = false)
    private BigDecimal maxPrice;
    @Positive  // > 0
    private Integer count;

    @PositiveOrZero  // >= 0
    private Integer stock;

    @Negative  // < 0
    private Integer debit;

    @NegativeOrZero  // <= 0
    private Integer credit;
}
```

Details:

- Type must be numeric
- Decimal versions for precision
- Convenience annotations available

@Email

Purpose: Email format validation.

Usage: String fields containing email.

Example:

```
public class ContactDto {
    @Email(message = "Invalid email format")
    private String primaryEmail;

    @Email(regexp = ".*@mycompany\\.com$", message = "Must be compa
ny email")
    private String workEmail;

    @Pattern(regexp = "^[A-Za-z0-9+_.-]+@(.+)$")
    private String customEmail;
}
```

Details:

- Basic RFC compliance
- Custom patterns possible
- Null values pass

@Pattern

Purpose: Regex pattern matching.

Usage: String validation with regex.

Example:

```
public class UserProfileDto {
    @Pattern(regexp = "^[A-Za-z0-9_]+$", message = "Username must be a
lphanumeric")
    private String username;

    @Pattern(regexp = "^\\+?[1-9]\\d{1,14}$", message = "Invalid phone num
ber")
    private String phoneNumber;

    @Pattern(
        regexp = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&+=])(?=\\S+
$).{8,}$",
        message = "Password must contain digit, lowercase, uppercase, speci
al character"
    )
    private String password;
}
```

Details:

- Java regex syntax
- Flags supported

- Complex validations possible

@Future, @Past, @PastOrPresent, @FutureOrPresent

Purpose: Temporal validation.

Usage: Date/time fields.

Example:

```
public class EventDto {
    @Future(message = "Event date must be in future")
    private LocalDateTime eventDate;

    @PastOrPresent(message = "Birth date cannot be in future")
    private LocalDate birthDate;

    @Past
    private Date createdDate;

    @FutureOrPresent
    private ZonedDateTime scheduledTime;
}
```

Details:

- Works with various date types
- Timezone aware
- Null values pass

Custom Validation Annotations

Purpose: Domain-specific validation.

Usage: Complex business rules.

Example:

```
// Custom annotation
@Target({ElementType.FIELD, ElementType.PARAMETER})
```

```
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PhoneNumberValidator.class)
public @interface ValidPhoneNumber {
    String message() default "Invalid phone number";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    String countryCode() default "US";
}

// Validator implementation
public class PhoneNumberValidator implements ConstraintValidator<ValidP
honeNumber, String> {
    private String countryCode;

    @Override
    public void initialize(ValidPhoneNumber annotation) {
        this.countryCode = annotation.countryCode();
    }

    @Override
    public boolean isValid(String phoneNumber, ConstraintValidatorContext c
ontext) {
        if (phoneNumber == null) {
            return true;  // Let @NotNull handle null
        }

        // Country-specific validation logic
        return validateForCountry(phoneNumber, countryCode);
    }
}

// Usage
public class ContactDto {
    @ValidPhoneNumber(countryCode = "US")
    private String phone;
}
```

Details:

- Reusable validation logic
- Access to annotation parameters
- Can add custom error messages

---

9. Caching Annotations

@EnableCaching

Purpose: Enables Spring's caching support.

Usage: Configuration class.

Example:

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();
        cacheManager.setCaches(Arrays.asList(
            new ConcurrentMapCache("users"),
            new ConcurrentMapCache("products")
        ));
        return cacheManager;
    }

    @Bean
    public KeyGenerator customKeyGenerator() {
        return (target, method, params) → {
            return method.getName() + "_" +
                Arrays.toString(params);
        };
    }
}
```

Details:

- Required for cache annotations
- Multiple cache providers supported
- Custom configuration possible

@Cacheable

Purpose: Caches method return value.

Usage: Read operations.

Example:

```
@Service
public class UserService {

    @Cacheable("users")
    public User findById(Long id) {
        // This method executes only on cache miss
        return userRepository.findById(id).orElse(null);
    }

    @Cacheable(
        value = "users",
        key = "#email.toLowerCase()",
        condition = "#email.length() > 5",
        unless = "#result == null"
    )
    public User findByEmail(String email) {
        return userRepository.findByEmail(email);
    }

    @Cacheable(
        value = "userStats",
        keyGenerator = "customKeyGenerator",
        sync = true  // Synchronized to prevent cache stampede
    )
    public UserStatistics calculateStats(Long userId, DateRange range) {
        return expensiveCalculation(userId, range);
    }
}
```

Details:

- Skip execution on cache hit
- Conditional caching supported
- Custom key generation

@CachePut

Purpose: Updates cache with method result.

Usage: Update operations.

Example:

```
@Service
public class ProductService {

    @CachePut(value = "products", key = "#product.id")
    public Product updateProduct(Product product) {
        // Always executes and updates cache
        return productRepository.save(product);
    }

    @CachePut(
        value = "products",
        condition = "#product.price > 100",
        unless = "#result.discontinued"
    )
    public Product saveProduct(Product product) {
        // Conditional cache update
        return productRepository.save(product);
    }
}
```

Details:

- Always executes method
- Updates cache with result
- Useful for write-through

@CacheEvict

Purpose: Removes entries from cache.

Usage: Delete operations or cache clearing.

Example:

```
@Service
public class CacheService {

    @CacheEvict(value = "users", key = "#id")
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }

    @CacheEvict(value = "products", allEntries = true)
    public void clearProductCache() {
        // Removes all entries from products cache
    }

    @CacheEvict(
        value = {"users", "userStats"},
        key = "#user.id",
        beforeInvocation = true  // Evict before method execution
    )
    public void riskyOperation(User user) {
        // Cache cleared even if method fails
    }

    @Scheduled(cron = "0 0 * * * *")  // Every hour
    @CacheEvict(value = "tempData", allEntries = true)
    public void scheduledCacheClear() {
        log.info("Cleared temporary cache");
    }
}
```

Details:

- Can clear entire cache
- Multiple caches supported
- Pre/post invocation options

@Caching

Purpose: Groups multiple cache operations.

Usage: Complex caching scenarios.

Example:

```
@Service
public class ComplexCacheService {

    @Caching(
        cacheable = {
            @Cacheable(value = "users", key = "#id"),
            @Cacheable(value = "userDetails", key = "#id")
        },
        put = {
            @CachePut(value = "recentUsers", key = "#id")
        }
    )
    public User getUser(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    @Caching(
        evict = {
            @CacheEvict(value = "users", key = "#user.id"),
            @CacheEvict(value = "userStats", key = "#user.id"),
            @CacheEvict(value = "userPosts", allEntries = true)
        }
    )
    public void updateUserAndEvictCaches(User user) {
        userRepository.save(user);
    }
}
```

Details:

- Combines multiple operations
- Executes in order
- Cleaner than multiple annotations

@CacheConfig

Purpose: Class-level cache configuration.

Usage: Share cache config across methods.

Example:

```
@Service
@CacheConfig(cacheNames = "users", keyGenerator = "customKeyGenera
tor")
public class UserCacheService {

    @Cacheable  // Uses "users" cache and custom key generator
    public User findById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    @CachePut(key = "#user.id")  // Overrides only key
    public User save(User user) {
        return userRepository.save(user);
    }

    @CacheEvict(allEntries = true)
    public void clearCache() {
        // Clears "users" cache
    }
}
```

Details:

- Reduces duplication
- Method level overrides class level
- Cleaner code

---

10. Messaging Annotations

RabbitMQ Annotations

@EnableRabbit

Purpose: Enables RabbitMQ listener annotations.

Usage: Configuration class.

Example:

```
@Configuration
@EnableRabbit
public class RabbitConfig {

    @Bean
    public Queue orderQueue() {
        return new Queue("orders", true);  // Durable queue
    }

    @Bean
    public TopicExchange orderExchange() {
        return new TopicExchange("order-exchange");
    }

    @Bean
    public Binding binding(Queue queue, TopicExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange).with("order.#");
    }

    @Bean
    public MessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }
}
```

Details:

- Enables @RabbitListener
- Auto-configuration available
- Custom converters supported

@RabbitListener

Purpose: Marks method as message listener.

Usage: Message consumer methods.

Example:

```
@Component
public class OrderListener {

    @RabbitListener(queues = "orders")
    public void processOrder(Order order) {
        // Process single message
        log.info("Received order: {}", order);
    }

    @RabbitListener(
        bindings = @QueueBinding(
            value = @Queue(value = "priority-orders", durable = "true"),
            exchange = @Exchange(value = "orders", type = "topic"),
            key = "order.priority.*"
        ),
        concurrency = "3-10",
        ackMode = "MANUAL"
    )
    public void processPriorityOrder(Order order, Channel channel,
                                    @Header(AmqpHeaders.DELIVERY_TAG) long tag) {
        try {
            // Process message
            channel.basicAck(tag, false);
        } catch (Exception e) {
            channel.basicNack(tag, false, true);  // Requeue
        }
    }

    @RabbitListener(queues = "notifications")
    @SendTo("notification-replies")  // Reply to another queue
    public NotificationResponse sendNotification(NotificationRequest reques
t) {
        // Process and return response
        return new NotificationResponse("Sent");
    }
}
```

Details:

- Auto-creates queues/bindings
- Supports manual acknowledgment
- Error handling options

@RabbitHandler

Purpose: Handles different message types.

Usage: Multiple handlers in one class.

Example:

```
@Component
@RabbitListener(queues = "multi-type-queue")
public class MultiTypeListener {

    @RabbitHandler
    public void handleOrder(Order order) {
        log.info("Processing order: {}", order);
    }

    @RabbitHandler
    public void handlePayment(Payment payment) {
        log.info("Processing payment: {}", payment);
    }

    @RabbitHandler(isDefault = true)
    public void handleDefault(Object object) {
        log.warn("Unknown message type: {}", object.getClass());
    }
}
```

Details:

- Type-based routing
- Default handler supported
- Clean message handling

Kafka Annotations

@EnableKafka

Purpose: Enables Kafka listener annotations.

Usage: Configuration class.

Example:

```
@Configuration
@EnableKafka
public class KafkaConfig {

    @Bean
    public ConsumerFactory<String, Object> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhos
t:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, Strin
gDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, Js
onDeserializer.class);
        props.put(JsonDeserializer.TRUSTED_PACKAGES, "*");
        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object> kafkaL
istenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3);
        factory.setCommonErrorHandler(new DefaultErrorHandler(
            new FixedBackOff(1000L, 3)
        ));
        return factory;
    }
}
```

Details:

- Enables @KafkaListener
- Configurable error handling
- Batch processing support

@KafkaListener

Purpose: Kafka message consumer.

Usage: Topic subscription.

Example:

```java
@Component
public class KafkaConsumer {

    @KafkaListener(topics = "orders", groupId = "order-service")
    public void consumeOrder(Order order) {
        log.info("Consumed order: {}", order);
    }

    @KafkaListener(
        topics = "events",
        containerFactory = "batchFactory",
        errorHandler = "customErrorHandler"
    )
    public void consumeBatch(List<Event> events) {
        log.info("Processing {} events", events.size());
        events.forEach(this::processEvent);
    }

    @KafkaListener(
        topicPartitions = @TopicPartition(
            topic = "users",
            partitionOffsets = {
                @PartitionOffset(partition = "0", initialOffset = "0"),
                @PartitionOffset(partition = "1", initialOffset = "100")
            }
        )
    )
    public void consumeFromSpecificPartition(User user) {
        // Consume from specific partitions and offsets
    }

    @KafkaListener(topics = "requests")
    @SendTo("responses")  // Send result to another topic
    public Response processRequest(Request request) {
        return new Response(request.getId(), "Processed");
    }
}
```

Details:

- Multiple topics supported
- Partition assignment
- Reply templates

@KafkaHandler

Purpose: Type-based message routing.

Usage: Multiple message types per topic.

Example:

```
@Component
@KafkaListener(topics = "domain-events", groupId = "event-processor")
public class EventProcessor {

    @KafkaHandler
    public void handleUserEvent(UserEvent event) {
        log.info("User event: {}", event);
    }

    @KafkaHandler
    public void handleOrderEvent(OrderEvent event) {
        log.info("Order event: {}", event);
    }

    @KafkaHandler(isDefault = true)
    public void handleUnknown(Object event) {
        log.warn("Unknown event type: {}", event.getClass());
    }
}
```

Details:

- Clean event handling
- Type safety
- Default handler

11. Configuration Annotations

@Configuration

Purpose: Marks class as configuration source.

Usage: Java-based configuration.

Example:

```
@Configuration
@ComponentScan(basePackages = "com.example.services")
@Import({DataConfig.class, SecurityConfig.class})
@ImportResource("classpath:legacy-config.xml")  // Import XML config
public class AppConfig {

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder
            .setConnectTimeout(Duration.ofSeconds(5))
            .setReadTimeout(Duration.ofSeconds(5))
            .build();
    }

    @Bean
    @Profile("production")
    public DataSource productionDataSource() {
        return DataSourceBuilder.create()
            .url("jdbc:postgresql://prod-server/db")
            .build();
    }

    @Bean
    @ConditionalOnMissingBean
    public ObjectMapper objectMapper() {
        return new ObjectMapper()
            .registerModule(new JavaTimeModule())
            .disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
    }
}
```

Details:

- Replacement for XML config
- Can import other configs
- Processed at startup

@Value

Purpose: Injects property values.

Usage: Field or method parameter injection.

Example:

```
@Component
public class AppProperties {

    @Value("${app.name:MyApp}")  // Default value
    private String appName;

    @Value("${app.timeout:30}")
    private int timeout;

    @Value("${app.features}")
    private List<String> features;  // Comma-separated list

    @Value("#{${app.settings}}")  // SpEL for map
    private Map<String, String> settings;

    @Value("#{systemProperties['user.home']}")
    private String userHome;

    @Value("#{@someBean.someMethod()}")
    private String fromBean;

    @Value("classpath:data/sample.json")
    private Resource sampleFile;

    @Value("${random.int(1,100)}")  // Random value
    private int randomNumber;
}
```

Details:

- Supports SpEL
- Type conversion
- Default values

@PropertySource

Purpose: Loads properties files.

Usage: Additional property sources.

Example:

```
@Configuration
@PropertySource("classpath:application.properties")
@PropertySource(value = "classpath:custom.properties", ignoreResourceN
otFound = true)
@PropertySources({
    @PropertySource("classpath:db.properties"),
    @PropertySource("file:${user.home}/app.properties")
})
public class PropertiesConfig {

    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyConfigurer
() {
        PropertySourcesPlaceholderConfigurer configurer =
            new PropertySourcesPlaceholderConfigurer();
        configurer.setIgnoreUnresolvablePlaceholders(true);
        configurer.setOrder(Ordered.LOWEST_PRECEDENCE);
        return configurer;
    }
}
```

Details:

- Multiple sources supported
- Order matters
- Can ignore missing files

@Profile

Purpose: Conditional bean/config activation.

Usage: Environment-specific configuration.

Example:

```
@Configuration
public class DatabaseConfig {

    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .build();
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:postgresql://prod-server/db");
        config.setMaximumPoolSize(20);
        return new HikariDataSource(config);
    }

    @Bean
    @Profile("!test")  // Not test
    public CacheManager cacheManager() {
        return new CaffeineCacheManager();
    }

    @Component
    @Profile({"dev", "test"})  // Multiple profiles
    public class MockEmailService implements EmailService {
        // Mock implementation for dev/test
    }
}
```

Details:

- Activated by spring.profiles.active
- Supports NOT operator
- Can combine profiles

@Import

Purpose: Imports configuration classes.

Usage: Modular configuration.

Example:

```
@Configuration
@Import({
    DataSourceConfig.class,
    SecurityConfig.class,
    CacheConfig.class
})
public class MainConfig {
    // Main configuration
}

// Conditional imports
@Configuration
@Import(DatabaseConfigurationSelector.class)
public class ConditionalConfig {
}

public class DatabaseConfigurationSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata metadata) {
        String dbType = System.getProperty("db.type", "mysql");
        return switch (dbType) {
            case "postgres" → new String[]{PostgresConfig.class.getName()};
            case "mongo" → new String[]{MongoConfig.class.getName()};
            default → new String[]{MysqlConfig.class.getName()};
        };
    }
}
```

Details:

- Modular configuration
- Dynamic imports possible
- ImportSelector for conditional

@ComponentScan

Purpose: Configures component scanning.

Usage: Specify packages to scan.

Example:

```
@Configuration
@ComponentScan(
    basePackages = {"com.example.services", "com.example.repositories"},
    basePackageClasses = {MarkerInterface.class},  // Type-safe package r
eference
    includeFilters = @ComponentScan.Filter(
        type = FilterType.ANNOTATION,
        classes = CustomComponent.class
    ),
    excludeFilters = {
        @ComponentScan.Filter(type = FilterType.REGEX, pattern = ".*Test.
*"),
        @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value =
LegacyService.class)
    },
    lazyInit = true,
    nameGenerator = CustomBeanNameGenerator.class
)
public class ScanConfig {
    // Configuration
}
```

Details:

- Multiple packages supported
- Custom filters
- Lazy initialization option

12. AOP (Aspect-Oriented Programming) Annotations

@EnableAspectJAutoProxy

Purpose: Enables AspectJ support.

Usage: Configuration class.

Example:

```
@Configuration
@EnableAspectJAutoProxy(
    proxyTargetClass = true,  // Use CGLIB proxies
    exposeProxy = true  // Allow self-invocation
)
public class AopConfig {

    @Bean
    public LoggingAspect loggingAspect() {
        return new LoggingAspect();
    }
}
```

Details:

- Required for @Aspect
- JDK vs CGLIB proxies
- Self-invocation support

@Aspect

Purpose: Marks class as aspect.

Usage: Cross-cutting concerns.

Example:

```
@Aspect
@Component
@Order(1)  // Aspect precedence
public class LoggingAspect {

    private static final Logger logger = LoggerFactory.getLogger(LoggingAs
pect.class);

    @Pointcut("@annotation(Loggable)")
    public void loggableMethods() {}

    @Pointcut("within(@org.springframework.stereotype.Repository *)")
    public void repositoryMethods() {}

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

Details:

- Contains advice methods
- Requires @EnableAspectJAutoProxy

- Can be ordered

@Before

Purpose: Advice executed before method.

Usage: Pre-processing logic.

Example:

```
@Aspect
@Component
public class SecurityAspect {

    @Before("@annotation(secured)")
    public void checkSecurity(JoinPoint joinPoint, Secured secured) {
        String[] roles = secured.value();
        // Security check logic
        if (!hasRequiredRole(roles)) {
            throw new AccessDeniedException("Insufficient privileges");
        }
    }

    @Before("execution(* com.example.service.*.save*(..)) && args(entit
y,..)")
    public void validateBeforeSave(JoinPoint joinPoint, Object entity) {
        log.info("Saving entity: {}", entity.getClass().getSimpleName());
        validateEntity(entity);
    }
}
```

Details:

- Access to method arguments
- Can throw exceptions
- Runs before target method

@After

Purpose: Advice executed after method.

Usage: Cleanup or logging.

Example:

```
@Aspect
@Component
public class ResourceCleanupAspect {

    @After("@annotation(CleanupRequired)")
    public void cleanup(JoinPoint joinPoint) {
        // Always executes, even if exception thrown
        cleanupResources();
    }

    @After("execution(* com.example.repository.*.find*(..))")
    public void logAfterFind() {
        MDC.clear();  // Clear logging context
    }
}
```

Details:

- Always executes
- No access to return value
- Like finally block

@AfterReturning

Purpose: Advice after successful execution.

Usage: Post-processing results.

Example:

```
@Aspect
@Component
public class AuditAspect {
    @AfterReturning(
        pointcut = "@annotation(Auditable)",
        returning = "result"
    )
    public void auditSuccess(JoinPoint joinPoint, Object result) {
        String method = joinPoint.getSignature().getName();
        Object[] args = joinPoint.getArgs();

        AuditLog log = new AuditLog();
        log.setMethod(method);
        log.setArgs(Arrays.toString(args));
        log.setResult(result.toString());
        log.setTimestamp(LocalDateTime.now());

        auditService.save(log);
    }

    @AfterReturning(
        pointcut = "execution(* com.example.service.*.create*(..))",
        returning = "entity"
    )
    public void logCreatedEntity(Object entity) {
        if (entity instanceof BaseEntity) {
            log.info("Created entity with ID: {}", ((BaseEntity) entity).getId());
        }
    }
}
```

Details:

- Access to return value
- Only on successful execution
- Can't modify return value

@AfterThrowing

Purpose: Advice after exception.

Usage: Exception handling/logging.

Example:

```
@Aspect
@Component
public class ExceptionHandlingAspect {

    @AfterThrowing(
        pointcut = "execution(* com.example.service.*.*(..))",
        throwing = "exception"
    )
    public void logException(JoinPoint joinPoint, Exception exception) {
        String method = joinPoint.getSignature().toShortString();
        log.error("Exception in method: {} with message: {}",
                  method, exception.getMessage());

        // Send alert for critical exceptions
        if (exception instanceof CriticalException) {
            alertingService.sendAlert(method, exception);
        }
    }

    @AfterThrowing(
        pointcut = "@annotation(Retriable)",
        throwing = "ex"
    )
    public void retryOnException(JoinPoint joinPoint, Exception ex) {
        // Could implement retry logic here
        retryManager.scheduleRetry(joinPoint, ex);
    }
}
```

Details:

- Only on exception
- Can't suppress exception
- Useful for logging

@Around

Purpose: Wraps method execution.

Usage: Full control over execution.

Example:

```
@Aspect
@Component
public class PerformanceAspect {

   @Around("@annotation(Timed)")
   public Object measureExecutionTime(ProceedingJoinPoint joinPoint) thr
ows Throwable {
      long start = System.currentTimeMillis();

      try {
         // Proceed with method execution
         Object result = joinPoint.proceed();
         return result;
      } finally {
         long duration = System.currentTimeMillis() - start;
         log.info("{} executed in {} ms",
               joinPoint.getSignature().toShortString(), duration);
      }
   }

   @Around("@annotation(cacheable)")
   public Object cache(ProceedingJoinPoint joinPoint, Cacheable cacheabl
e) throws Throwable {
      String key = generateKey(joinPoint, cacheable);

      // Check cache
      Object cached = cacheManager.get(key);
      if (cached != null) {
         log.debug("Cache hit for key: {}", key);
         return cached;
      }
```

```
        // Execute method
        Object result = joinPoint.proceed();

        // Store in cache
        if (result != null) {
            cacheManager.put(key, result, cacheable.ttl());
        }

        return result;
    }

    @Around("@annotation(Retry)")
    public Object retryOperation(ProceedingJoinPoint joinPoint) throws Thro
wable {
        int attempts = 0;
        Exception lastException;

        do {
            try {
                return joinPoint.proceed();
            } catch (Exception e) {
                lastException = e;
                attempts++;
                if (attempts < 3) {
                    Thread.sleep(1000 * attempts);  // Exponential backoff
                }
            }
        } while (attempts < 3);

        throw lastException;
    }
}
```

Details:

- Complete control
- Can modify arguments/result

- Must call proceed()

@Pointcut

Purpose: Defines reusable pointcut.

Usage: Pointcut expression definition.

Example:

```
@Aspect
@Component
public class CommonPointcuts {

    // Annotation-based
    @Pointcut("@annotation(org.springframework.transaction.annotation.Tra
nsactional)")
    public void transactionalMethods() {}

    // Package-based
    @Pointcut("within(com.example.service..*)")
    public void inServiceLayer() {}

    // Method pattern
    @Pointcut("execution(public * com.example..*.*(..))")
    public void publicMethods() {}

    // Bean name pattern
    @Pointcut("bean(*Service)")
    public void serviceBeans() {}

    // Combining pointcuts
    @Pointcut("inServiceLayer() && publicMethods()")
    public void publicServiceMethods() {}

    // With parameters
    @Pointcut("execution(* com.example..*.find*(..)) && args(id,..)")
    public void findMethods(Long id) {}
    // Custom annotation with parameter
    @Pointcut("@annotation(performanceTracking)")
    public void performanceTracked(PerformanceTracking performanceTrac
king) {}
}

// Usage in advice
@Before("CommonPointcuts.transactionalMethods()")
public void beforeTransaction() {
    // Advice logic
}
```

Details:

- Reusable expressions
- Can be combined
- Improves maintainability

---

13. Scheduling Annotations

@EnableScheduling

Purpose: Enables scheduled task execution.

Usage: Configuration class.

Example:

```
@Configuration
@EnableScheduling
@ConditionalOnProperty(
    name = "scheduling.enabled",
    havingValue = "true",
    matchIfMissing = true
)
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler
();
        scheduler.setPoolSize(10);
        scheduler.setThreadNamePrefix("scheduled-");
        scheduler.setAwaitTerminationSeconds(60);
        scheduler.setWaitForTasksToCompleteOnShutdown(true);
        scheduler.initialize();

        taskRegistrar.setTaskScheduler(scheduler);
    }

    @Bean
    public TaskScheduler taskScheduler() {
        return new ConcurrentTaskScheduler();
    }
}
```

Details:

- Required for @Scheduled
- Configurable thread pool
- Can be conditional

@Scheduled

Purpose: Marks method for scheduled execution.

Usage: Periodic task execution.

Example:

```
@Component
@ConditionalOnProperty(name = "batch.jobs.enabled", havingValue = "tru
e")
public class BatchJobs {

    // Fixed delay - waits after completion
    @Scheduled(fixedDelay = 5000)  // 5 seconds
    public void processQueue() {
        log.info("Processing queue items");
        // Task logic
    }

    // Fixed rate - runs every interval
    @Scheduled(fixedRate = 60000, initialDelay = 10000)  // Every minute, 1
0s initial delay
    public void syncData() {
        log.info("Syncing data");
        // Sync logic
    }

    // Cron expression
    @Scheduled(cron = "0 0 2 * * ?")  // Every day at 2 AM
    public void dailyCleanup() {
        log.info("Running daily cleanup");
        // Cleanup logic
    }

    // Cron with zone
    @Scheduled(cron = "0 0 9 * * MON-FRI", zone = "America/New_York")
    public void businessHoursTask() {
        log.info("Business hours task");
    }

    // Dynamic scheduling from properties
    @Scheduled(fixedDelayString = "${batch.process.delay:5000}")
    public void configuredTask() {
        // Task with configurable delay
    }

    // Complex cron from properties
    @Scheduled(cron = "${batch.report.cron:0 0 6 * * ?}")
    public void generateReports() {
        try {
            reportService.generateDailyReports();
        } catch (Exception e) {
            log.error("Report generation failed", e);
            // Scheduled tasks should handle their own exceptions
        }
    }
}
```

Details:

- Multiple scheduling options
- Property-driven configuration
- Timezone support

@Async

Purpose: Enables asynchronous execution.

Usage: Non-blocking method execution.

Example:

```
@Configuration
@EnableAsync
public class AsyncConfig implements AsyncConfigurer {

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("async-");
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.Caller
RunsPolicy());
        executor.initialize();
        return executor;
    }

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHa
ndler() {
        return (throwable, method, params) → {
            log.error("Async method {} threw exception", method.getName(), thr
owable);
        };
    }
}

@Service
public class AsyncService {

    @Async
    public void processInBackground(String data) {
        log.info("Processing in thread: {}", Thread.currentThread().getName
());
        // Long running task
    }

    @Async("customExecutor")  // Use specific executor
    public CompletableFuture<String> asyncWithResult(String input) {
        try {
            Thread.sleep(1000);
            return CompletableFuture.completedFuture("Processed: " + input);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return CompletableFuture.failedFuture(e);
        }
    }

    @Async
    public Future<User> findUser(Long id) {
        User user = userRepository.findById(id).orElse(null);
        return new AsyncResult<>(user);
    }
}
```

Details:

- Requires @EnableAsync

- Returns Future/CompletableFuture
- Custom executors supported

---

14. Conditional Annotations

@Conditional

Purpose: Base conditional annotation.

Usage: Custom condition evaluation.

Example:

```
// Custom condition
public class OnDatabaseTypeCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetad
ata metadata) {
        String dbType = context.getEnvironment().getProperty("database.typ
e");
        Map<String, Object> attributes = metadata.getAnnotationAttributes(
            ConditionalOnDatabaseType.class.getName());
        String expectedType = (String) attributes.get("value");
        return expectedType.equalsIgnoreCase(dbType);
    }
}

// Custom annotation
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Conditional(OnDatabaseTypeCondition.class)
public @interface ConditionalOnDatabaseType {
    String value();
}

// Usage
@Configuration
public class DatabaseConfig {
    @Bean
    @ConditionalOnDatabaseType("mysql")
    public DataSource mysqlDataSource() {
        return new MysqlDataSource();
    }

    @Bean
    @ConditionalOnDatabaseType("postgres")
    public DataSource postgresDataSource() {
        return new PostgresDataSource();
    }
}
```

Details:

- Foundation for other conditionals
- Custom conditions possible
- Flexible evaluation

Common Conditional Annotations

Purpose: Built-in conditional checks.

Usage: Conditional bean creation.

Example:

```
@Configuration
public class ConditionalConfig {

    // Property conditions
    @Bean
    @ConditionalOnProperty(
        prefix = "feature",
        name = "advanced-search",
        havingValue = "true",
                matchIfMissing = false
    )
    public SearchService advancedSearchService() {
        return new AdvancedSearchService();
    }

    // Class presence
    @Bean
    @ConditionalOnClass(name = "redis.clients.jedis.Jedis")
    public CacheManager redisCacheManager() {
        return new RedisCacheManager();
    }

    // Bean presence/absence
    @Bean
    @ConditionalOnMissingBean(CacheManager.class)
    public CacheManager defaultCacheManager() {
        return new ConcurrentMapCacheManager();
    }

    // Expression
    @Bean
    @ConditionalOnExpression("${cache.type:'none'} == 'redis' && ${cach
e.cluster.enabled:false}")
    public RedisClusterConfiguration redisClusterConfig() {
        return new RedisClusterConfiguration();
    }

    // Resource existence
    @Bean
    @ConditionalOnResource(resources = "classpath:custom-config.xml")
    public CustomConfiguration customConfig() {
        return new CustomConfiguration();
    }

    // Web application
    @Configuration
    @ConditionalOnWebApplication(type = ConditionalOnWebApplication.Ty
pe.SERVLET)
    public static class WebConfig {
        @Bean
        public WebMvcConfigurer corsConfigurer() {
            return new WebMvcConfigurer() {
                @Override
                public void addCorsMappings(CorsRegistry registry) {
                    registry.addMapping("/**").allowedOrigins("*");
                }
            };
        }
    }

    // Profile
    @Bean
    @Profile("!production")  // Can combine with conditionals
    @ConditionalOnProperty(name = "debug.enabled", havingValue = "tru
e")
    public DebugService debugService() {
        return new DebugService();
    }

    // Java version
    @Bean
    @ConditionalOnJava(JavaVersion.ELEVEN)
    public ModernFeatureService modernService() {
        return new ModernFeatureService();
    }

    // JNDI
    @Bean
    @ConditionalOnJndi("java:comp/env/jdbc/myDataSource")
    public DataSource jndiDataSource() {
        return new JndiDataSourceLookup().getDataSource("java:comp/env/jd
bc/myDataSource");
    }
}
```

Details:

- Rich set of conditions

- Combine multiple conditions
- Auto-configuration foundation

---

15. Spring Boot Actuator Annotations

@Endpoint

Purpose: Creates custom actuator endpoint.

Usage: Management and monitoring endpoints.

Example:

```
@Component
@Endpoint(id = "custom-health")
public class CustomHealthEndpoint {

    private final HealthService healthService;

    @ReadOperation
    public CustomHealth health() {
        return CustomHealth.builder()
            .status(healthService.getStatus())
            .details(healthService.getDetails())
            .build();
    }

    @WriteOperation
    public void updateHealth(String status) {
        healthService.updateStatus(status);
    }

    @DeleteOperation
    public void resetHealth() {
        healthService.reset();
    }
}
```

```
@Component
@WebEndpoint(id = "features")
public class FeatureEndpoint {

    @ReadOperation
    public WebEndpointResponse<Map<String, Boolean>> features() {
        Map<String, Boolean> features = featureService.getAllFeatures();
        return new WebEndpointResponse<>(features, 200);
    }

    @ReadOperation
    public WebEndpointResponse<Boolean> getFeature(@Selector String na
me) {
        Boolean enabled = featureService.isEnabled(name);
        if (enabled == null) {
            return new WebEndpointResponse<>(404);
        }
        return new WebEndpointResponse<>(enabled, 200);
    }
}
```

Details:

- Custom management endpoints
- RESTful operations
- Integration with Spring Security

@EndpointWebExtension

Purpose: Extends existing endpoint for web.

Usage: Add web-specific functionality.

Example:

```
@Component
@EndpointWebExtension(endpoint = InfoEndpoint.class)
public class CustomInfoEndpointWebExtension {

    @ReadOperation
    public WebEndpointResponse<Map<String, Object>> info() {
        Map<String, Object> info = new HashMap<>();
        info.put("custom", "Custom web info");
        info.put("timestamp", Instant.now());
        return new WebEndpointResponse<>(info);
    }
}
```

Details:

- Enhance existing endpoints
- Web-specific responses
- Additional functionality

---

16. Reactive Annotations

@EnableWebFlux

Purpose: Enables Spring WebFlux.

Usage: Reactive web applications.

Example:

```
@Configuration
@EnableWebFlux
public class WebFluxConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer confi
gurer) {
        configurer.defaultCodecs().maxInMemorySize(10 * 1024 * 1024);
    }

    @Bean
    public RouterFunction<ServerResponse> routes() {
        return RouterFunctions
```

```
            .route(GET("/api/stream"), this::streamData)
            .andRoute(POST("/api/process"), this::processData);
    }
}
```

Details:

- Non-blocking web stack
- Netty server by default
- Functional routing

Reactive Repository Annotations

Purpose: Reactive data access.

Usage: Non-blocking database operations.

Example:

```
@Repository
public interface ReactiveUserRepository extends ReactiveCrudRepository<
User, String> {

    @Query("SELECT * FROM users WHERE email = :email")
    Mono<User> findByEmail(String email);

    @Query("SELECT * FROM users WHERE age > :age")
    Flux<User> findByAgeGreaterThan(int age);

    @Modifying
    @Query("UPDATE users SET last_login = :date WHERE id = :id")
    Mono<Integer> updateLastLogin(String id, LocalDateTime date);
}

@Service
public class ReactiveUserService {

    @Autowired
    private ReactiveUserRepository repository;
    public Flux<User> getAllUsers() {
        return repository.findAll()
            .delayElements(Duration.ofMillis(100))  // Simulate slow stream
            .timeout(Duration.ofSeconds(5))
            .onErrorResume(TimeoutException.class, e -> Flux.empty());
    }

    public Mono<User> createUser(Mono<User> userMono) {
        return userMono
            .flatMap(repository::save)
            .doOnSuccess(user -> log.info("Created user: {}", user.getId()))
            .doOnError(error -> log.error("Failed to create user", error));
    }
}
```

Details:

- Returns Mono/Flux
- Non-blocking operations
- Backpressure support

---

17. Monitoring and Metrics Annotations

@Timed

Purpose: Records method execution time.

Usage: Performance monitoring.

Example:

```
@RestController
public class MetricsController {

    @Timed(
        value = "user.get",
        description = "Time taken to fetch user",
        percentiles = {0.5, 0.95, 0.99},
        histogram = true
    )
    @GetMapping("/users/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.findById(id);
    }

    @Timed(value = "db.query", longTask = true)
    public List<User> complexQuery() {
        // Long running query
        return userRepository.complexQuery();
    }
}

// Enable @Timed aspect
@Configuration
@EnableAspectJAutoProxy
public class MetricsConfig {
    @Bean
    public TimedAspect timedAspect(MeterRegistry registry) {
        return new TimedAspect(registry);
    }
}
```

Details:

- Micrometer integration
- Custom metrics
- Percentile tracking

@Counted

Purpose: Counts method invocations.

Usage: Track operation frequency.

Example:

```
@Service
public class PaymentService {

    @Counted(value = "payment.processed", description = "Number of pay
ments processed")
    public PaymentResult processPayment(PaymentRequest request) {
        // Process payment
        return paymentGateway.process(request);
    }

    @Counted(value = "payment.failed", recordFailuresOnly = true)
    public void riskyOperation() {
        // Only counts when exception is thrown
    }
}
```

Details:

- Simple counter metric
- Success/failure tracking
- Tags support

---

18. OpenAPI/Swagger Annotations

@Operation

Purpose: Describes API operation.

Usage: API documentation.

Example:

```
@RestController
@Tag(name = "User Management", description = "Operations related to us
ers")
public class UserApiController {

    @Operation(
    summary = "Get user by ID",
    description = "Fetches a user by their unique identifier",
    tags = {"users"},
    responses = {
        @ApiResponse(
            responseCode = "200",
            description = "User found",
            content = @Content(
                mediaType = "application/json",
                schema = @Schema(implementation = User.class)
            )
        ),
        @ApiResponse(
            responseCode = "404",
            description = "User not found",
            content = @Content(
                mediaType = "application/json",
                schema = @Schema(implementation = ErrorResponse.class)
            )
        )
    }
)
@GetMapping("/users/{id}")
public User getUser(
    @Parameter(description = "User ID", required = true, example = "123")
    @PathVariable Long id
) {
    return userService.findById(id);
}

@Operation(summary = "Create new user")
@PostMapping("/users")
public User createUser(
    @RequestBody
    @Schema(description = "User creation request")
    UserCreateRequest request
) {
    return userService.create(request);
    }
}

@Schema(description = "User entity")
public class User {
    @Schema(description = "Unique identifier", example = "123")
    private Long id;

    @Schema(description = "User email", example = "user@example.com", r
equired = true)
    @Email
    private String email;

    @Schema(description = "User roles", allowableValues = {"ADMIN", "USE
R", "GUEST"})
    private List<String> roles;
}
```

Details:

- OpenAPI 3.0 specification
- Rich documentation
- Interactive UI with Swagger

---

19. Spring Integration Annotations

@IntegrationComponentScan

Purpose: Enables Spring Integration.

Usage: Message-driven architecture.

Example:

```
@Configuration
@EnableIntegration
@IntegrationComponentScan
public class IntegrationConfig {
    @Bean
    public MessageChannel inputChannel() {
        return MessageChannels.direct().get();
    }

    @Bean
    public MessageChannel outputChannel() {
        return MessageChannels.publishSubscribe().get();
    }
}
```

@MessagingGateway

Purpose: Creates messaging gateway interface.

Usage: Simplified messaging API.

Example:

```
@MessagingGateway(defaultRequestChannel = "inputChannel")
public interface OrderGateway {

    @Gateway(requestChannel = "orderChannel", replyTimeout = 5000)
    OrderResult processOrder(Order order);

    @Gateway(requestChannel = "asyncChannel")
    Future<OrderResult> processOrderAsync(Order order);
}

@Component
public class OrderProcessor {

    @ServiceActivator(inputChannel = "orderChannel")
    public OrderResult handle(Order order) {
        // Process order
        return new OrderResult(order.getId(), "PROCESSED");
    }
    @Transformer(inputChannel = "transformChannel", outputChannel = "ou
tputChannel")
    public OrderDto transform(Order order) {
        return new OrderDto(order);
    }

    @Filter(inputChannel = "filterChannel", outputChannel = "validOrderCha
nnel")
    public boolean filterValidOrders(Order order) {
        return order.getAmount() > 0 && order.getItems().size() > 0;
    }
}
```

Details:

- Enterprise Integration Patterns
- Message routing and transformation
- Async processing support

20. Best Practices and Common Patterns

Annotation Composition

Purpose: Create custom composed annotations.

Usage: Reduce annotation boilerplate.

Example:

```
// Custom composed annotation
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Transactional
@Timed
@PreAuthorize("hasRole('ADMIN')")
public @interface AdminOperation {
    @AliasFor(annotation = Timed.class, attribute = "value")
    String metricName() default "";

    @AliasFor(annotation = Transactional.class, attribute = "readOnly")
    boolean readOnly() default false;
}

// Usage
@Service
public class AdminService {
    @AdminOperation(metricName = "admin.delete", readOnly = false)
    public void deleteAllData() {
        // Admin operation with transaction, timing, and security
    }
}

// REST endpoint composition
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@GetMapping
@ResponseStatus(HttpStatus.OK)
@Operation(summary = "Get resource")
public @interface GetResource {
    @AliasFor(annotation = GetMapping.class, attribute = "value")
    String[] path() default {};
}
```

Meta-Annotations

Purpose: Annotations that annotate other annotations.

Usage: Framework development.

Example:

```
// Custom qualifier annotation
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
```

```
public @interface DatabaseType {
    String value();
}

// Custom validation annotation
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = StrongPasswordValidator.class)
@Documented
public @interface StrongPassword {
    String message() default "Password is not strong enough";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    int minLength() default 8;
    boolean requireUppercase() default true;
    boolean requireLowercase() default true;
    boolean requireDigit() default true;
    boolean requireSpecialChar() default true;
}
```

Common Anti-Patterns to Avoid

1. **Over-annotation:** Don't add unnecessary annotations

   ```
   // Bad - redundant annotations
   @Component  // Already implied by @Service
   @Service
   @Transactional  // Better at method level
   public class UserService { }

   // Good
   @Service
   public class UserService {
       @Transactional
       public void updateUser(User user) { }
   }
   ```

2. **Wrong annotation placement**

   ```
   // Bad - @Transactional on private method (won't work)
   @Service
   public class Service {
       @Transactional
       private void privateMethod() { }  // No proxy!
   }

   // Good
   @Service
   public class Service {
       @Transactional
       public void publicMethod() { }
   }
   ```

3. **Circular dependencies with field injection**

   ```
   // Bad - circular dependency
   @Service
   public class ServiceA {
       @Autowired
       private ServiceB serviceB;
   }

   @Service
   public class ServiceB {
       @Autowired
       private ServiceA serviceA;
   }

   // Good - constructor injection with @Lazy
   @Service
   public class ServiceA {
       private final ServiceB serviceB;

       public ServiceA(@Lazy ServiceB serviceB) {
           this.serviceB = serviceB;
   ```

```
    }
}
```

Quick Reference - Annotation Categories

Core Container

- `@Component`, `@Service`, `@Repository`, `@Controller`
- `@Configuration`, `@Bean`, `@Scope`
- `@Autowired`, `@Qualifier`, `@Primary`, `@Lazy`
- `@PostConstruct`, `@PreDestroy`

Web Layer

- `@RestController`, `@RequestMapping`, `@GetMapping`, `@PostMapping`
- `@RequestParam`, `@PathVariable`, `@RequestBody`, `@ResponseBody`
- `@ResponseStatus`, `@ExceptionHandler`, `@ControllerAdvice`
- `@CrossOrigin`, `@SessionAttributes`, `@ModelAttribute`

Data Access

- `@Entity`, `@Table`, `@Id`, `@GeneratedValue`
- `@Column`, `@JoinColumn`, `@OneToMany`, `@ManyToOne`
- `@Query`, `@Modifying`, `@Transactional`
- `@Repository`, `@EntityGraph`, `@NamedQuery`

Security

- `@EnableWebSecurity`, `@EnableMethodSecurity`
- `@PreAuthorize`, `@PostAuthorize`, `@Secured`
- `@AuthenticationPrincipal`, `@RolesAllowed`

Testing

- `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`
- `@MockBean`, `@SpyBean`, `@TestConfiguration`

- `@DirtiesContext`, `@ActiveProfiles`, `@TestPropertySource`

Configuration

- `@Value`, `@ConfigurationProperties`, `@PropertySource`
- `@Profile`, `@Conditional`, `@ConditionalOnProperty`
- `@Import`, `@ComponentScan`, `@EnableAutoConfiguration`

Async & Scheduling

- `@EnableAsync`, `@Async`, `@EnableScheduling`, `@Scheduled`

Caching

- `@EnableCaching`, `@Cacheable`, `@CachePut`, `@CacheEvict`
- `@Caching`, `@CacheConfig`

Messaging

- `@EnableRabbit`, `@RabbitListener`, `@RabbitHandler`
- `@EnableKafka`, `@KafkaListener`, `@KafkaHandler`
- `@JmsListener`, `@SendTo`

Cloud

- `@EnableEurekaClient`, `@FeignClient`, `@LoadBalanced`
- `@EnableConfigServer`, `@RefreshScope`
- `@CircuitBreaker`, `@Retry`, `@RateLimiter`

Validation

- `@Valid`, `@Validated`, `@NotNull`, `@NotEmpty`, `@NotBlank`
- `@Size`, `@Min`, `@Max`, `@Email`, `@Pattern`
- `@Past`, `@Future`, `@AssertTrue`, `@AssertFalse`

AOP

- `@EnableAspectJAutoProxy`, `@Aspect`, `@Pointcut`
- `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`, `@Around`

This comprehensive cheat sheet covers all major Spring Boot annotations with detailed explanations, usage examples, and important details. Each annotation includes its purpose, typical usage scenarios, code examples, and key points to remember. Save this as a reference for your Spring Boot development work!

✈️ Happy Coding!
