**Spring Boot Annotations Reference Guide**

**Library Management System Project**

This document catalogs all Spring Boot and testing annotations used in the Library Management System project, explaining their purpose and showing where they appear in the codebase.

**Core Spring Framework Annotations**

**@SpringBootApplication**

**Purpose**: The main annotation that combines three key annotations to bootstrap a Spring Boot application.

**What it does**:

* @Configuration: Marks the class as a configuration class
* @EnableAutoConfiguration: Enables Spring Boot's auto-configuration
* @ComponentScan: Scans the package and subpackages for Spring components

**Used in**:

* LibraryManagementSystemApplication.java (main application class)

**Example**:

@SpringBootApplication

public class LibraryManagementSystemApplication {

public static void main(String[] args) {

SpringApplication.run(LibraryManagementSystemApplication.class, args);

}

}

**Dependency Injection Annotations**

**@Component**

**Purpose**: Marks a class as a Spring-managed component that should be auto-detected during component scanning.

**What it does**:

* Creates a singleton bean in the Spring application context
* Allows the class to be injected into other components
* Generic stereotype annotation for any Spring-managed component

**Used in**:

* AuthHelper.java - Utility class for authentication operations
* CacheHelper.java - Utility class for caching operations
* ValidationHelper.java - Utility class for validation logic

**Example**:

@Component

public class AuthHelper {

// Spring will create and manage this instance

}

**@Service**

**Purpose**: Specialized @Component annotation for service layer classes containing business logic.

**What it does**:

* Marks the class as a service layer component
* Indicates that the class contains business logic
* Creates a Spring-managed singleton bean

**Used in**:

* AuthService.java - Authentication business logic
* BookService.java - Book management business logic
* UserService.java - User management business logic
* RentalService.java - Rental operations business logic

**Example**:

@Service

public class AuthService {

// Business logic for user authentication

}

**@Repository**

**Purpose**: Specialized @Component annotation for data access layer classes.

**What it does**:

* Marks the class as a data access component
* Enables automatic exception translation for database exceptions
* Creates a Spring-managed singleton bean

**Used in**:

* JsonUserRepository.java - User data access implementation
* JsonBookRepository.java - Book data access implementation
* JsonRentalRepository.java - Rental data access implementation

**Example**:

@Repository

public class JsonUserRepository implements UserRepository {

// Data access logic for users

}

**@Configuration**

**Purpose**: Marks a class as a source of bean definitions and configuration.

**What it does**:

* Indicates that the class contains @Bean methods
* Allows programmatic configuration of Spring beans
* Replaces XML configuration files

**Used in**:

* SecurityConfig.java - Security configuration and bean definitions

**Example**:

@Configuration

public class SecurityConfig {

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder(10);

}

}

**@Bean**

**Purpose**: Marks a method as a producer of a bean to be managed by Spring.

**What it does**:

* Tells Spring to call this method and manage the returned object
* The returned object becomes a Spring-managed bean
* Used inside @Configuration classes

**Used in**:

* SecurityConfig.java - Creates SecurityFilterChain and PasswordEncoder beans

**Example**:

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder(10);

}

**Web Layer Annotations**

**@RestController**

**Purpose**: Combines @Controller and @ResponseBody for REST API endpoints.

**What it does**:

* Marks the class as a web controller
* Automatically serializes return values to JSON
* Handles HTTP requests and returns data (not views)

**Used in**:

* AuthController.java - Authentication endpoints
* BookController.java - Book management endpoints
* UserController.java - User management endpoints
* AdminController.java - Admin-only endpoints
* HealthController.java - Health check endpoint

**Example**:

@RestController

@RequestMapping("/auth")

public class AuthController {

// REST API endpoints for authentication

}

**@RequestMapping**

**Purpose**: Maps HTTP requests to handler methods or classes.

**What it does**:

* Defines the base URL path for all endpoints in a controller
* Can specify HTTP methods, headers, and other request conditions

**Used in**:

* AuthController.java - Maps to /auth path
* BookController.java - Maps to /books path
* UserController.java - Maps to /users path
* AdminController.java - Maps to /admin path

**Example**:

@RestController

@RequestMapping("/books")

public class BookController {

// All endpoints will start with /books

}

**@GetMapping**

**Purpose**: Shortcut for @RequestMapping(method = RequestMethod.GET).

**What it does**:

* Maps HTTP GET requests to specific handler methods
* Used for retrieving data

**Used in**:

* BookController.java - searchBooks(), getMyRentals()
* UserController.java - getMyProfile(), getAllUsers(), getMyRentalHistory()
* AdminController.java - exportBooks()
* HealthController.java - health()

**Example**:

@GetMapping("/me")

public ResponseEntity<?> getMyProfile(HttpServletRequest request) {

// Handle GET /users/me

}

**@PostMapping**

**Purpose**: Shortcut for @RequestMapping(method = RequestMethod.POST).

**What it does**:

* Maps HTTP POST requests to specific handler methods
* Used for creating data or performing actions

**Used in**:

* AuthController.java - login(), register(), logout()
* BookController.java - addBook(), rentBook(), returnBook()
* UserController.java - promoteUser(), demoteUser()
* AdminController.java - importBooks(), addBook()

**Example**:

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody LoginRequest request) {

// Handle POST /auth/login

}

**@PutMapping**

**Purpose**: Shortcut for @RequestMapping(method = RequestMethod.PUT).

**What it does**:

* Maps HTTP PUT requests to specific handler methods
* Used for updating existing data

**Used in**:

* UserController.java - updateMyProfile(), updateUser()
* AdminController.java - updateBook()

**Example**:

@PutMapping("/me")

public ResponseEntity<?> updateMyProfile(@RequestBody Map<String, String> updateData) {

// Handle PUT /users/me

}

**@DeleteMapping**

**Purpose**: Shortcut for @RequestMapping(method = RequestMethod.DELETE).

**What it does**:

* Maps HTTP DELETE requests to specific handler methods
* Used for deleting data

**Used in**:

* UserController.java - deleteUser()
* AdminController.java - deleteBook()

**Example**:

@DeleteMapping("/{id}")

public ResponseEntity<?> deleteUser(@PathVariable String id) {

// Handle DELETE /users/{id}

}

**@RequestBody**

**Purpose**: Maps the HTTP request body to a method parameter.

**What it does**:

* Automatically deserializes JSON request body to Java objects
* Uses Jackson for JSON conversion

**Used in**:

* AuthController.java - Converting JSON to LoginRequest, RegisterRequest
* BookController.java - Converting JSON to book data maps
* UserController.java - Converting JSON to update data maps
* AdminController.java - Converting JSON to book data maps

**Example**:

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody LoginRequest request) {

// request is automatically populated from JSON

}

**@PathVariable**

**Purpose**: Extracts values from the URL path and maps them to method parameters.

**What it does**:

* Captures dynamic parts of URLs (like IDs)
* Automatically converts string values to appropriate types

**Used in**:

* BookController.java - rentBook(), returnBook()
* UserController.java - updateUser(), deleteUser(), promoteUser(), demoteUser()
* AdminController.java - updateBook(), deleteBook()

**Example**:

@PostMapping("/{id}/rent")

public ResponseEntity<?> rentBook(@PathVariable String id) {

// id is extracted from URL like /books/123/rent

}

**@RequestParam**

**Purpose**: Maps HTTP request parameters to method parameters.

**What it does**:

* Extracts query parameters from URLs
* Can specify default values and whether parameters are required

**Used in**:

* BookController.java - searchBooks() for pagination and search filters
* UserController.java - getAllUsers(), getMyRentalHistory() for pagination
* AdminController.java - importBooks() for file upload

**Example**:

@GetMapping

public ResponseEntity<?> searchBooks(

@RequestParam(required = false) String title,

@RequestParam(defaultValue = "0") int page) {

// title comes from ?title=... (optional)

// page comes from ?page=... (defaults to 0)

}

**Testing Annotations**

**@ExtendWith(MockitoExtension.class)**

**Purpose**: Integrates Mockito with JUnit 5 for unit testing.

**What it does**:

* Enables Mockito annotations like @Mock and @InjectMocks
* Automatically initializes mocks before each test
* Manages mock lifecycle

**Used in**:

* AuthServiceTest.java - Unit testing with mocked dependencies
* BookServiceTest.java - Unit testing with mocked dependencies

**Example**:

@ExtendWith(MockitoExtension.class)

class AuthServiceTest {

// Mockito annotations will work here

}

**@Mock**

**Purpose**: Creates mock objects for dependencies in unit tests.

**What it does**:

* Creates a fake implementation of an interface or class
* Allows you to control what methods return during tests
* Isolates the class being tested from its dependencies

**Used in**:

* AuthServiceTest.java - Mocking UserRepository, ValidationHelper, PasswordEncoder
* BookServiceTest.java - Mocking BookRepository, RentalRepository

**Example**:

@Mock

private UserRepository userRepository;

// This creates a fake UserRepository for testing

**@InjectMocks**

**Purpose**: Creates an instance of the class being tested and injects mocked dependencies.

**What it does**:

* Creates the actual object you want to test
* Automatically injects all @Mock objects into it
* Uses constructor injection, setter injection, or field injection

**Used in**:

* AuthServiceTest.java - Creates AuthService with injected mocks
* BookServiceTest.java - Creates BookService with injected mocks

**Example**:

@InjectMocks

private AuthService authService;

// Creates AuthService and injects mocked dependencies

**@BeforeEach**

**Purpose**: Marks a method to run before each test method.

**What it does**:

* Sets up test data and conditions
* Runs before every @Test method
* Used for test initialization

**Used in**:

* AuthServiceTest.java - Sets up test user data
* BookServiceTest.java - Sets up test book data
* EndToEndWorkflowTest.java - Initializes services and repositories
* LibraryIntegrationTest.java - Creates test files and services

**Example**:

@BeforeEach

void setUp() {

// This runs before each test method

}

**@Test**

**Purpose**: Marks a method as a test method.

**What it does**:

* Tells JUnit this method should be executed as a test
* Test methods should contain assertions to verify behavior
* Tests should be independent and repeatable

**Used in**:

* All test classes - marking individual test methods

**Example**:

@Test

void testLoginSuccess() {

// Test logic with assertions

}

**@TempDir**

**Purpose**: Creates a temporary directory for file-based tests.

**What it does**:

* JUnit creates a temporary directory before the test
* Directory is automatically cleaned up after the test
* Useful for testing file operations

**Used in**:

* LibraryIntegrationTest.java - Creating temporary JSON files for testing

**Example**:

@TempDir

Path tempDir;

// JUnit creates and manages this temporary directory

**API Documentation Annotations**

**@Tag**

**Purpose**: Groups related API endpoints in Swagger/OpenAPI documentation.

**What it does**:

* Organizes endpoints into logical groups in Swagger UI
* Provides descriptions for endpoint groups
* Improves API documentation structure

**Used in**:

* AuthController.java - Groups authentication endpoints
* BookController.java - Groups book-related endpoints
* UserController.java - Groups user management endpoints
* AdminController.java - Groups admin-only endpoints

**Example**:

@Tag(name = "Authentication", description = "User authentication and session management")

@RestController

public class AuthController {

// All endpoints grouped under "Authentication" in Swagger

}

**@Operation**

**Purpose**: Documents individual API endpoints with descriptions and metadata.

**What it does**:

* Provides summary and description for each endpoint
* Documents what the endpoint does
* Improves API documentation readability

**Used in**:

* All controller classes - documenting individual endpoint methods

**Example**:

@Operation(

summary = "User login",

description = "Authenticate user credentials and create session"

)

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody LoginRequest request) {

// Endpoint documentation appears in Swagger UI

}

**@ApiResponses and @ApiResponse**

**Purpose**: Documents possible HTTP response codes and their meanings.

**What it does**:

* Lists all possible response codes (200, 400, 401, etc.)
* Explains what each response code means
* Helps API consumers understand possible outcomes

**Used in**:

* All controller classes - documenting response codes for each endpoint

**Example**:

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Login successful"),

@ApiResponse(responseCode = "401", description = "Invalid credentials"),

@ApiResponse(responseCode = "500", description = "Internal server error")

})

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody LoginRequest request) {

// Documents what response codes this endpoint can return

}

**@Parameter**

**Purpose**: Documents method parameters in API documentation.

**What it does**:

* Describes what each parameter does
* Specifies if parameters are optional or required
* Provides examples for parameter usage

**Used in**:

* All controller classes - documenting path variables and request parameters

**Example**:

@GetMapping

public ResponseEntity<?> searchBooks(

@Parameter(description = "Search by book title (optional)")

@RequestParam(required = false) String title) {

// Parameter is documented in Swagger UI

}

**Summary**

This project demonstrates key Spring Boot concepts through annotations:

**Dependency Injection**: @Component, @Service, @Repository create managed beans that Spring automatically wires together.

**Web Layer**: @RestController, @RequestMapping, @GetMapping, @PostMapping create REST API endpoints.

**Configuration**: @Configuration and @Bean set up application configuration and create custom beans.

**Testing**: @Mock, @InjectMocks, @Test enable comprehensive unit and integration testing.

**Documentation**: @Tag, @Operation, @ApiResponse create professional API documentation.

The annotation-driven approach reduces boilerplate code and makes the application structure clear and maintainable. Each annotation serves a specific purpose in the Spring Boot ecosystem, from dependency management to web request handling to testing infrastructure.