Complete AOP (Aspect-Oriented Programming) Spring Boot Notes

1. What is AOP?

- **Definition**: Aspect Oriented Programming (AOP) is a programming paradigm that aims to segregate cross-cutting functionalities from business logic
- **Purpose**: Modularizes cross-cutting concerns like logging, security, validation, and transaction management
- **Benefits**: Improves code maintainability, reduces code duplication, and separates concerns

2. Spring Boot Application Architecture

Three Main Layers:

- **Web Layer**: Exposes services using RESTful web services
- **Business Layer**: Handles business logic and core functionality
- **Data Layer**: Manages data persistence logic

Cross-Cutting Concerns:

- Logging
- Security
- Validation
- Transaction management
- Error handling
- Performance monitoring
- Caching

3. Core AOP Concepts

Key Terminology:

- **Aspect**: A module that encapsulates cross-cutting concerns
- **Join Point**: A point in program execution (method call, exception handling)
- **Pointcut**: Expression that defines where advice should be applied
- **Advice**: Code that runs at a particular join point
- **Weaving**: Process of applying aspects to target objects
- **Proxy**: AOP creates proxy objects to intercept method calls

4. Types of Advice in AspectJ AOP

@Before Advice

- **Execution**: Runs before the target method
- **Characteristics**:
    - Cannot prevent method execution (unless throws exception)

    - Used for validation, logging entry points
    - No access to method return value

```
@Before("execution(* com.example.service.*.*(..))")
public void beforeAdvice(JoinPoint joinPoint) {
    System.out.println("Method " + joinPoint.getSignature() + " started at " + new
Date());
}
```

@After Advice

- **Execution**: Runs after method completion (regardless of outcome)
- **Characteristics**:
    - Executes whether method returns normally or throws exception
    - Used for cleanup operations
    - Similar to finally block

```
@After("execution(* com.example.service.*.*(..))")
public void afterAdvice(JoinPoint joinPoint) {
    System.out.println("Method " + joinPoint.getSignature() + " ended at " + new
Date());
}
```

@AfterReturning Advice

- **Execution**: Runs after successful method completion
- **Characteristics**:
    - Only executes if method returns normally
    - Can access return value
    - Used for logging successful operations

```
@AfterReturning(value = "execution(*
com.example.service.EmployeeService.addEmployee(..))",
                returning = "employee")
public void afterReturningAdvice(JoinPoint joinPoint, Employee employee) {
    System.out.println("Employee saved successfully with ID: " +
employee.getId());
}
```

@AfterThrowing Advice

- **Execution**: Runs when method throws exception
- **Characteristics**:
    - Only executes on exception
    - Can access thrown exception

- Used for error logging and handling

```
@AfterThrowing(value = "execution(*
com.example.service.EmployeeService.addEmployee(..))",
               throwing = "exception")
public void afterThrowingAdvice(JoinPoint joinPoint, Exception exception) {
    System.out.println("Method threw exception: " + exception.getMessage());
}
```

@Around Advice

- **Execution**: Surrounds method execution
- **Characteristics**:
  - Most powerful advice type
  - Can modify method arguments and return values
  - Controls method execution via proceed()
  - Can prevent method execution

```
@Around("execution(* com.example.service.EmployeeService.addEmployee(..))")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("Around advice - before method execution");

    // Modify arguments if needed
    Object[] args = joinPoint.getArgs();

    try {
        Object result = joinPoint.proceed(args);
        System.out.println("Around advice - after successful execution");
        return result;
    } catch (Exception e) {
        System.out.println("Around advice - exception occurred");
        throw e;
    }
}
```

5. Pointcut Expressions

Common Patterns:

- `execution(* com.example.service.*.*(..))` - All methods in service package
- `execution(public * *(..))` - All public methods
- `execution(* com.example.service.UserService.save*(..))` - Methods starting with 'save'
- `@annotation(org.springframework.web.bind.annotation.GetMapping)` - Methods with specific annotation

Wildcards:

- `*` - Matches any single element
- `..` - Matches zero or more parameters or packages
- `+` - Matches subclasses/implementations

6. Advanced AOP Scenarios

Conditional Logging Based on Method Logic

```
@Around("execution(* com.example.service.*.*(..))")
public Object conditionalLogging(ProceedingJoinPoint joinPoint) throws Throwable {
    Object[] args = joinPoint.getArgs();

    // Check request type parameter
    if (args.length > 0 && args[0] instanceof Request) {
        Request req = (Request) args[0];
        if ("admin".equals(req.getReqType())) {
            System.out.println("Admin request logged");
        }
    }

    return joinPoint.proceed();
}
```

Modifying Request Parameters

```
@Around("execution(* com.example.service.processRequest(..))")
public Object modifyRequest(ProceedingJoinPoint joinPoint) throws Throwable {
    Object[] args = joinPoint.getArgs();

    // Modify specific parameters
    if (args[0] instanceof Request) {
        Request req = (Request) args[0];
        req.setReqType("MODIFIED");
    }

    return joinPoint.proceed(args);
}
```

Handling Multiple Return Types

```
@Around("execution(* com.example.service.*.*(..))")
public Object handleMultipleReturnTypes(ProceedingJoinPoint joinPoint) throws
Throwable {
    String methodName = joinPoint.getSignature().getName();
    Object result = joinPoint.proceed();

    // Handle different return types based on method
```

```
        if (methodName.startsWith("get")) {
            return result; // Return as-is for getter methods
        } else if (methodName.startsWith("save")) {
            // Wrap save operations in ResponseEntity
            return ResponseEntity.ok(result);
        }

        return result;
    }
```

7. Execution Order of Advice

When Multiple Advice Types Present:

1. **@Around** (before proceed())
2. **@Before**
3. **Target Method Execution**
4. **@Around** (after proceed())
5. **@After**
6. **@AfterReturning** (if success) OR **@AfterThrowing** (if exception)

Controlling Order with @Order Annotation:

```
@Aspect
@Order(1)
public class LoggingAspect { }

@Aspect
@Order(2)
public class SecurityAspect { }
```

8. Best Practices for AOP

Folder Structure:

```
src/main/java/
├── com/example/
│   ├── aspect/
│   │   ├── LoggingAspect.java
│   │   ├── SecurityAspect.java
│   │   └── PerformanceAspect.java
│   ├── service/
│   ├── controller/
│   └── repository/
```

Code Organization:

- **Separate Concerns**: One aspect per cross-cutting concern
- **Specific Pointcuts**: Use precise pointcut expressions
- **Minimal Performance Impact**: Keep advice lightweight
- **Proper Exception Handling**: Handle exceptions in advice appropriately

Configuration:

```
@Configuration
@EnableAspectJAutoProxy
public class AopConfig {
    // Configuration beans if needed
}
```

9. Common Use Cases

Performance Monitoring:

```
@Around("@annotation(Timed)")
public Object measureExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable
{
    long start = System.currentTimeMillis();
    Object result = joinPoint.proceed();
    long end = System.currentTimeMillis();
    System.out.println("Execution time: " + (end - start) + "ms");
    return result;
}
```

Caching:

```
@Around("@annotation(Cacheable)")
public Object cacheResult(ProceedingJoinPoint joinPoint) throws Throwable {
    String key = generateCacheKey(joinPoint);
    Object cached = cache.get(key);

    if (cached != null) {
        return cached;
    }

    Object result = joinPoint.proceed();
    cache.put(key, result);
    return result;
}
```

Audit Logging:

```
@AfterReturning("execution(* com.example.service.*Service.save*(..))")
public void auditSaveOperations(JoinPoint joinPoint) {
    String methodName = joinPoint.getSignature().getName();
    Object[] args = joinPoint.getArgs();
    // Log audit information
    auditService.logOperation(methodName, args);
}
```

10. Important Notes

Proxy Limitations:

- AOP works through proxies
- Internal method calls bypass AOP
- Only public methods can be intercepted (by default)
- Self-invocation doesn't trigger advice

Performance Considerations:

- Advice adds overhead to method calls
- Use specific pointcuts to minimize impact
- Avoid heavy operations in frequently called advice

Debugging Tips:

- Enable AOP debug logging
- Use @EnableAspectJAutoProxy(exposeProxy = true) if needed
- Test advice behavior thoroughly

Dependencies Required:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

11. Testing AOP

Unit Testing Aspects:

```
@Test
public void testLoggingAspect() {
    // Mock dependencies
    // Call advised method
    // Verify aspect behavior
}
```

Integration Testing:

```
@SpringBootTest
@TestMethodOrder(OrderAnnotation.class)
public class AopIntegrationTest {
    // Test complete AOP behavior in context
}
```