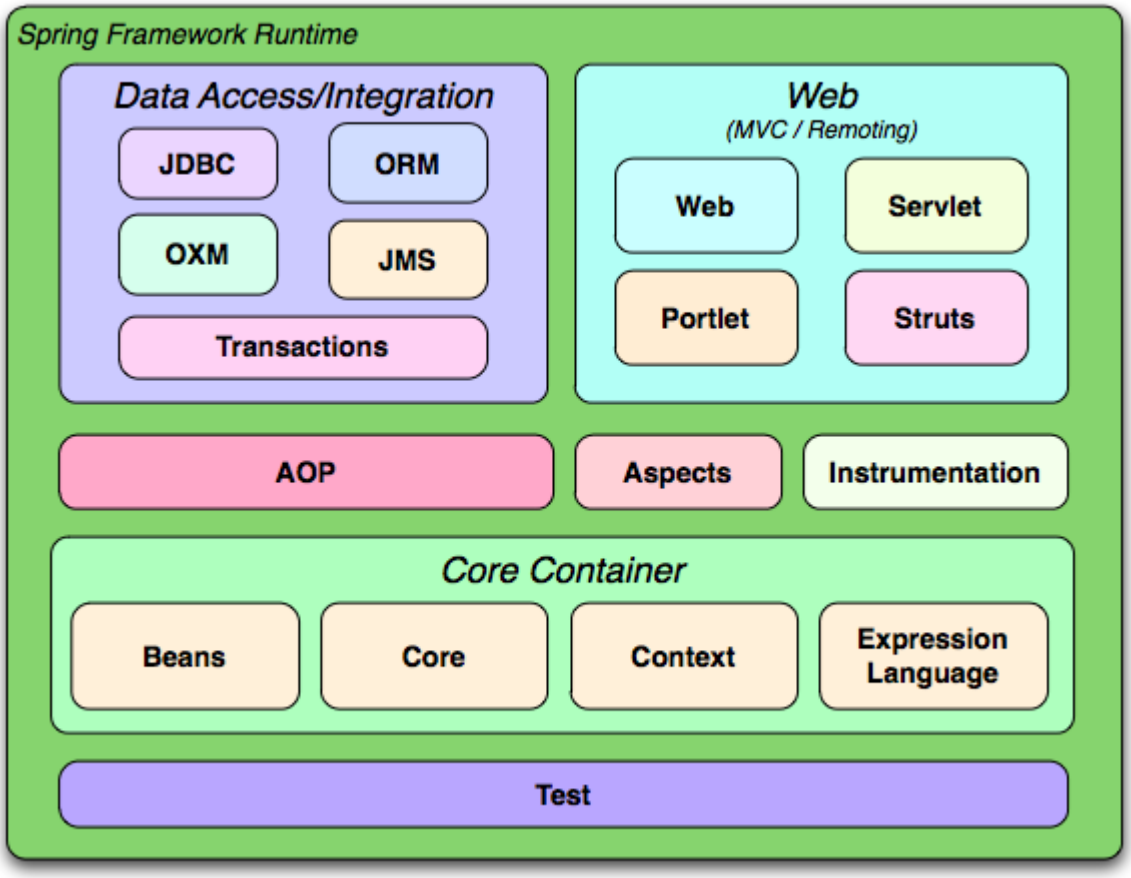
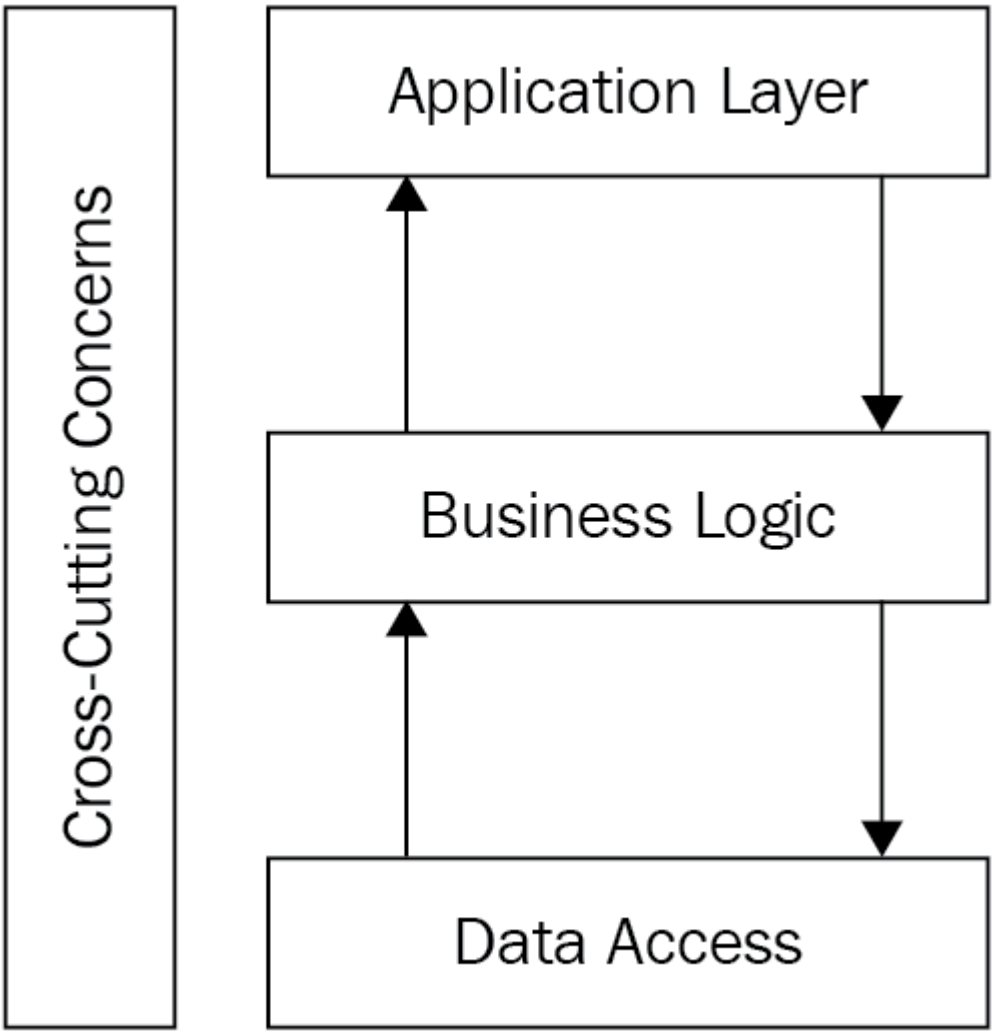
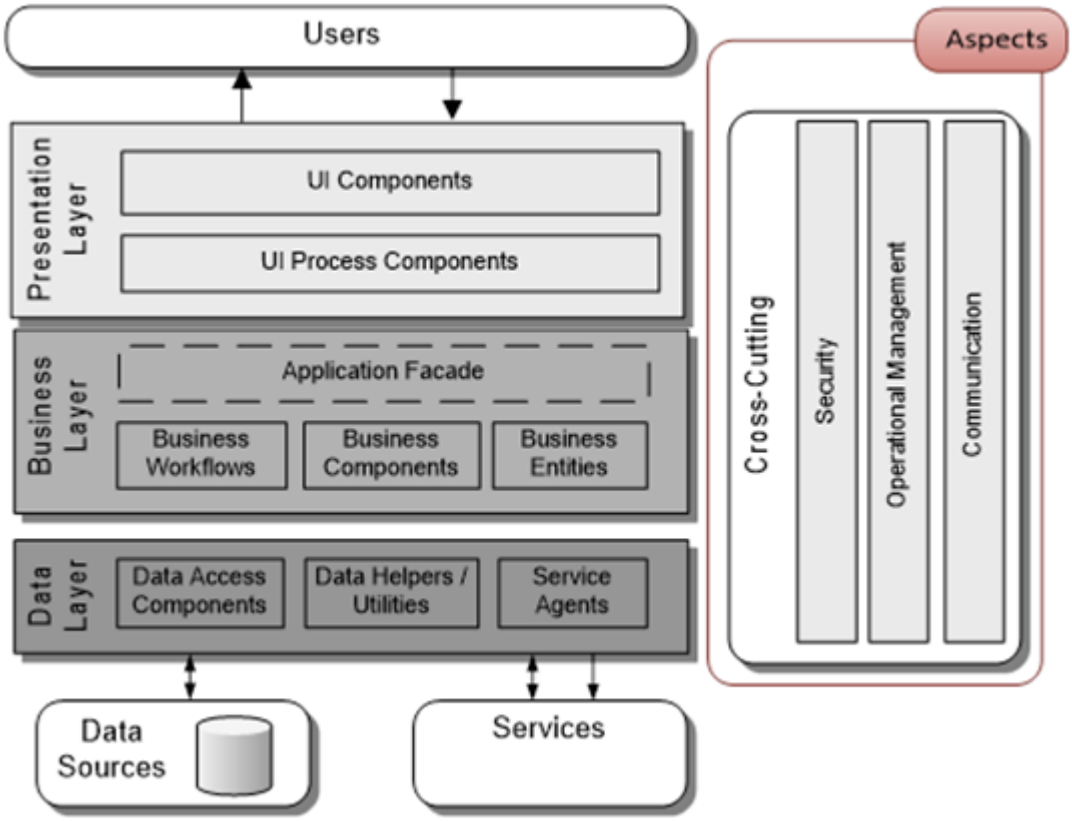
Spring AOP

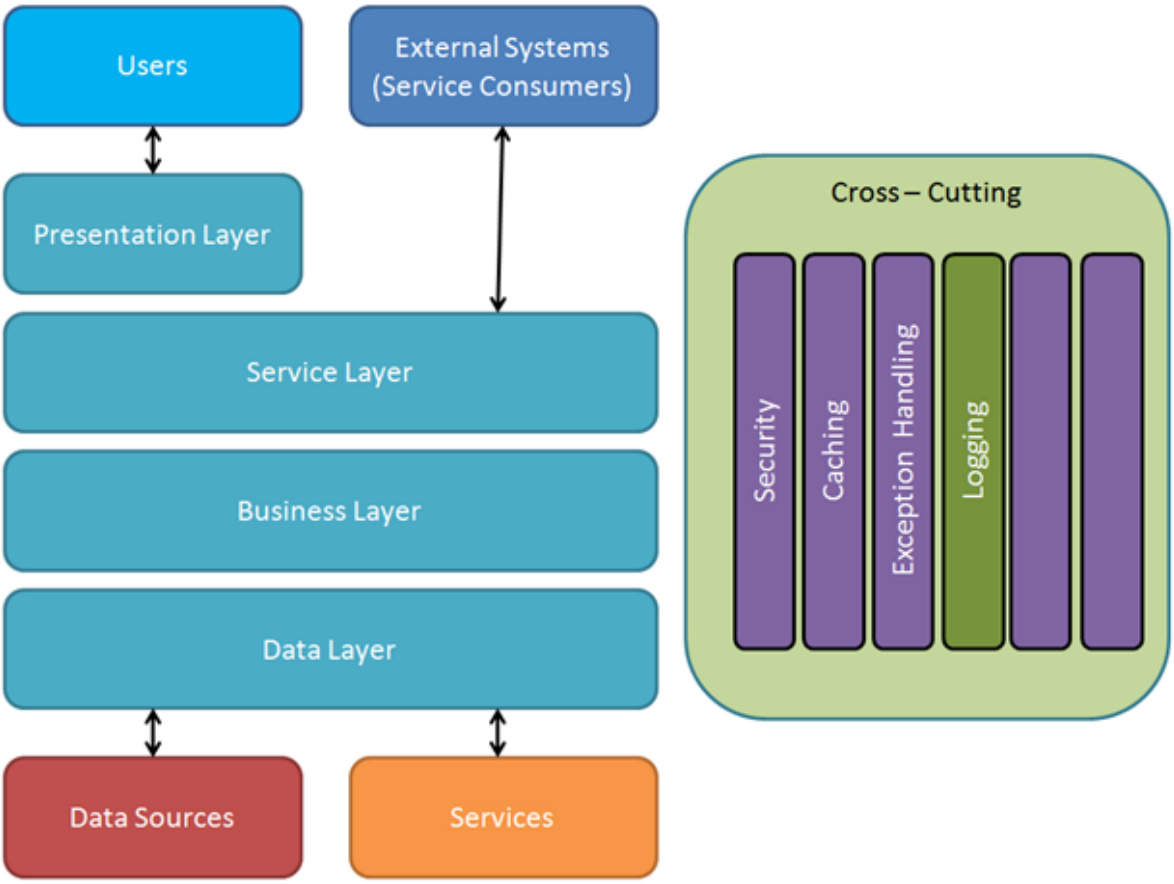


Introduction to AOP

Aspect Oriented Programming (AOP) compliments OOPs because it provides modularity. But, the key unit of modularity here is considered as an aspect rather than class. Here, AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.


Cross-Cutting Concerns
Application Layer
Business Logic
Data Access


Users
Aspects
Presentation Layer
UI Components
UI Process Components
Business Layer
Application Facade
Business Workflows
Business Components
Business Entities
Data Layer
Data Access Components
Data Helpers / Utilities
Service Agents
Cross-Cutting
Security
Operational Management
Communication
Data Sources
Services


Users
External Systems (Service Consumers)
Presentation Layer
Service Layer
Business Layer
Data Layer
Data Sources
Services
Cross – Cutting
Security
Caching
Exception Handling
Logging

A **cross-cutting concern** is the one that affects the whole application and should be centralized under the same block in the code, such as transaction management, authentication, logging, security etc.

As we know Spring AOP enables Aspect-Oriented Programming in spring applications. Here, aspects enable the modularization of concerns such as transaction management, logging or security that cut across multiple types and objects (often termed **crosscutting concerns**).

In this case, AOP provides a way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations.

By doing this, it makes easy to maintain code in the present and future as well.

So, you can add/remove c/concerns without recompiling complete source code simply by changing configuration files (if you are applying aspects suing XML configuration).

Why AOP?

The most important functionality is AOP provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods as shown in the figure:

```
class A{
1. public void m1(){...}
2. public void m2(){...}
3. public void m3(){...}
4. public void m4(){...}
5. public void m5(){...}
6. public void n1(){...}
7. public void n2(){...}
8. public void p1(){...}
9. public void p2(){...}
10. public void p3(){...}
11. }
12.
```

There are **5 methods that starts from m**, 2 methods that starts from n and 3 methods that starts from p.

Understanding Scenario I have to maintain log and send notification after calling methods that starts from m.

Problem without AOP We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification; you need to change all the methods. It leads to the maintenance problem.

Solution with AOP We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. **Its entry is given in the xml file or through annotation**

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

***The solution with AOP*– With AOP, we don't have to call methods from the method.**

We can simply define the additional concern like maintaining a log, sending notification etc.

Core AOP Concepts

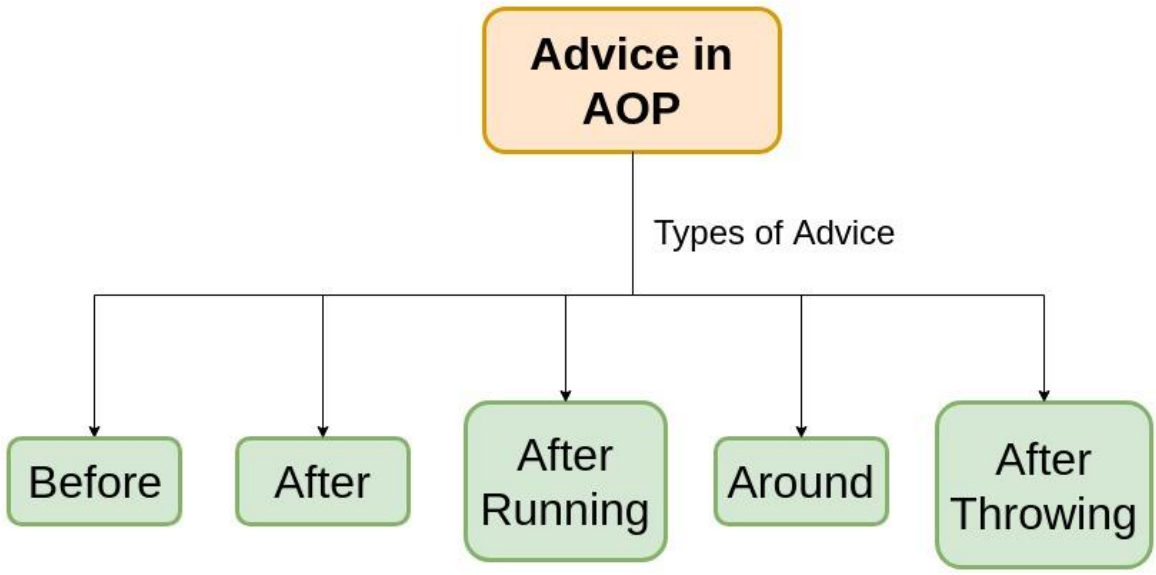
Spring AOP consists of 7 core concepts which are depicted in the following diagram:

1. **Aspect**: The *aspect* is nothing but a class that implements the JEE application concerns which cut through multiple classes, such as transaction management, security etc. Aspects can be a normal class configured through Spring XML configuration. It can also be regular classes annotated using **@Aspect annotation.**

2. **Joinpoint:** The *joinpoint* is a *candidate* point in the program execution where an aspect can be plugged in. It could be a method that is being called, an exception being thrown, or even a field being modified.
3. **Advice:** *Advice* are the specific actions taken for a particular joinpoint. Basically, they are the methods that get executed when a certain joinpoint meets a matching pointcut in the application.
4. **Pointcut:** A *Pointcut* is an expression that is matched with join points to determine whether advice needs to be executed or not.
5. **Target Object:** These are the objects on which advices are applied. In Spring AOP, a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.
6. **Proxy:** It is an object that is created after applying advice to the target object. In clients perspective, object, the target object, and the proxy object are same.
7. **Weaving:** *Weaving* is the process of linking an aspect with other application types or objects to create an advised object.

Spring AOP: Types of Advices

Several types of advices present in Spring AOP are as follows:



- **Before:** Here, advices execute before the joinpoint methods and are configured using ***@Before*** annotation mark.
- **After returning:** These advice types execute after the joinpoint methods complete the execution normally. They are configured using @AfterReturning annotation mark.
- **After throwing:** Here, advices execute only when the joinpoint method exits by throwing an exception. They are configured using @AfterThrowing annotation mark.
- **Around:** These advice types execute before and after a join point and are configured using @Around annotation mark.

AspectJ Concepts

The important aspect of Spring is the Aspect-Oriented Programming (AOP) framework. As we all know, the key unit of modularity in OOP(Object

Oriented Programming) is the class, similarly, in AOP the unit of modularity is the aspect.

Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. To implement these concerns, AspectJ comes into the picture. AspectJ, a compatible extension to the Java programming language, is one implementation of AOP.

It has grown into a complete and popular AOP framework. Since AspectJ annotations are supported by more and more AOP frameworks, AspectJ-style aspects are more likely to be reused in other AOP frameworks that support AspectJ.