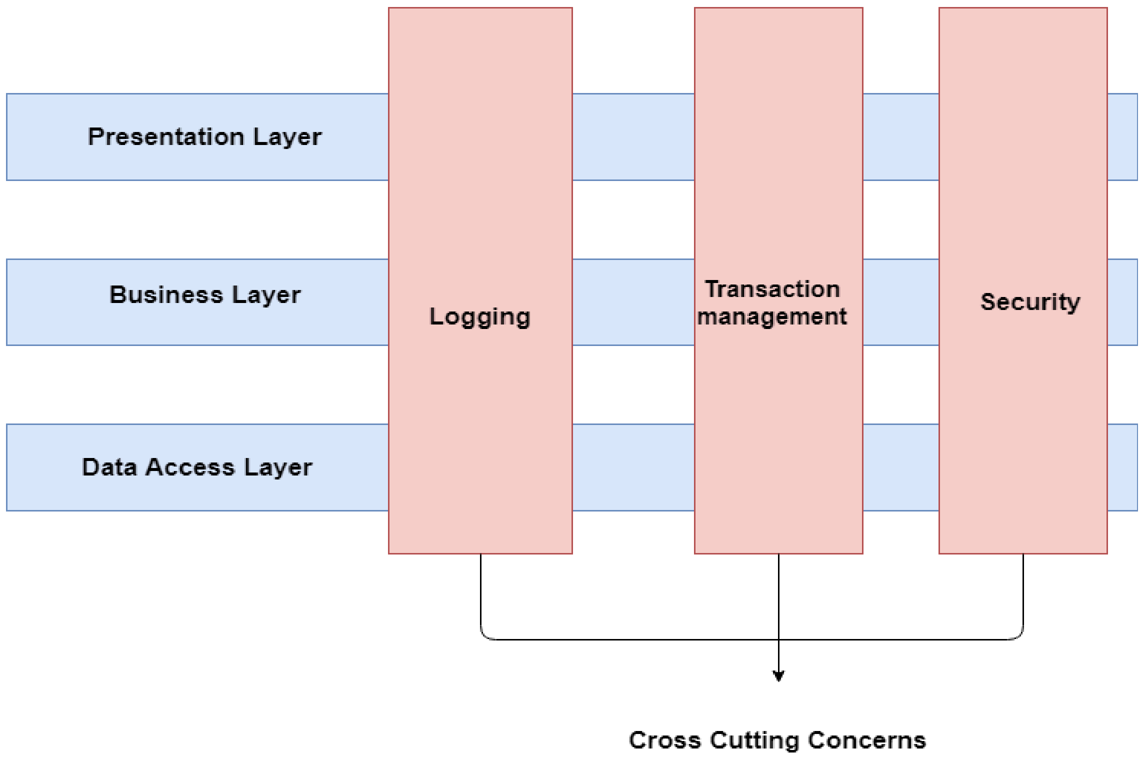
SPRING AOP

Spring AOP (Aspect-Oriented Programming) is a key feature of the **Spring Framework,** which provides a way to dynamically add additional behavior to methods or classes. **AOP** complements **Object-Oriented Programming (OOP)** by allowing the separation of concerns.

AOP framework is one of the key components of the Spring framework. Basically, we use this framework to enable **middleware services** with the spring application in a **loosely coupled** manner.

We are aware that each layer in a web application has **distinct** responsibilities. Additionally, there are other responsibilities like **validation, logging, security, caching,** etc. that are applied across many tiers. These are referred to as cross-cutting concerns here. Therefore, if we continue to implement these cross-cutting issues in each business logic independently, the amount of code will expand, leading to a more complex and challenging to manage code.



So to overcome this problem, Using AOP, we write codes for cross-cutting concerns separately, then we link it with the business logic i.e also called **weaving**. Thus, implementing **Modularity** on overall application, and making the codes easy to manage.

The **main aim of AOP** is to separate services from the actual business code. In simple words, we can add some more functionality to your business code without writing anything extra in your actual code.

Here's a breakdown of key concepts in Spring AOP:

Aspect is a module that implements cross-cutting concern by encapsulating the advice and pointcuts.

Advice is the actual code which is invoked during the method execution either before or after by Spring AOP framework.

Pointcuts is a point or a condition to execute aspects for business methods by linking the advice with the join points.

Join point is the possible position in your business method where the advice can be applied. This point can be the execution of a method or handling of an exception.

Example : Let's consider a simple logging aspect applied with and without using Spring AOP.

Without Applying AOP:
Suppose you have a **Calculator class** responsible for performing arithmetic operations, and you want to add logging functionality to log method calls without using AOP.

```
public class Calculator { public
   int add(int a, int b) {
      log("Adding " + a + " and " + b);
      return a + b;
   }

   public int subtract(int a, int b) {
      log("Subtracting " + b + " from " + a);
      return a - b;
   }

   private void log(String message) { System.out.println("Logging:
      " + message);
   }
}
```

In this the logging logic is directly mixed with the business logic of the Calculator class. The log method needs to be manually called from each method where logging is required. This can lead to **code duplication** and makes it harder to maintain, especially if the logging logic needs to be updated or changed.

With Applying AOP:
Now, let's see how the same logging functionality can be implemented using Spring AOP.

```
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

   @Before("execution(* Calculator.*(..))")
   public void logBeforeMethodCall() {
      System.out.println("Logging: Method is about to be executed");
   }
}
```

In this the logging logic is encapsulated within the LoggingAspect class, separating it from the Calculator class. We use a pointcut expression (@Before("execution(* Calculator.*(..))")) to specify where the advice (logging) should be applied. In this case, it applies to all methods of the Calculator class. We don't need to modify the Calculator class to add logging functionality, which keeps it clean and focused on its main purpose.

Comparison:

Without AOP: Logging logic is scattered throughout the application, leading to code duplication and mixing of concerns.
With AOP: Logging logic is centralized in a separate aspect, promoting better code organization, easier maintenance, and improved readability.

In **summary**, Spring AOP allows you to separate cross-cutting concerns (such as logging) from the main application logic, leading to cleaner and more maintainable code.