**Logging with Spring AOP**

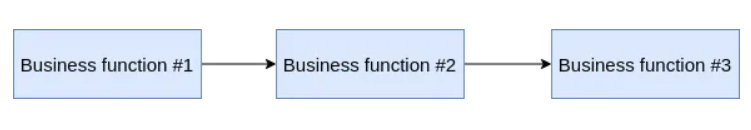
|  |  |  |
| --- | --- | --- |
| **S NO** | **Contents** | **Page No** |
| 1 | Introduction | 3 |
| 2 | Working | 3-5 |
| 3 | Methodology | 6-7 |
| 4 | Results | 8 |
| 5 | Conclusion | 9 |

# **Introduction**

# Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It means you can add additional behaviour without modifying the code itself. This is achieved with the help of Spring’s AOP framework.

# **Working**

For instance, we have an application that contains modules with some business logic.



But we need to add some handling of our results. Let’s take logging for example. Usually, we’d inject additional code at the end of every module.

A close-up of a sign

Description automatically generated

Some shared functionality like logging is called **cross-cutting concern**. Here are a few sources [(#1](https://stackoverflow.com/questions/23700540/cross-cutting-concern-example), [#2](https://stackoverflow.com/questions/23700540/cross-cutting-concern-example)).

But we’d like to use the AOP approach and separate our logging from business logic.

A diagram of a business function

Description automatically generated

Our new logging module will catch the output of other business modules in **join points**— points during the execution of a program, such as the execution of a method or the handling of an exception.

A diagram of a business function

Description automatically generated

To embed logging, we unify our join points into **pointcut**. For this, in Spring we use the expression language.

A diagram of a business function

Description automatically generated

For this, we use **advices —**actions at a particular *Join point*.

Types of advises:

* **Before**— before method execution.
* **After**— after method execution.
* **After returning** — after method returning.
* **After throwing** — after method throwing an exception.
* **After finally** — after method executing “finally” block.
* **Around** — allows us to add some preprocessing, postprocessing, or even bypass method execution.

A diagram of a point cut

Description automatically generated

**Aspect —**module which contains pointcuts and advises.

**Methodology**

Pre-requisites

Technologies: Spring’s AOP, Spring web, Spring Data JPA, Lombok, MySQL connector

**Step 1:**

Create a Spring starter project from STS and add required dependencies.

A screenshot of a computer

Description automatically generated

**Step 2:** And let’s create a new service and controller class  and write simple method like getting, posting and deleting data.

A computer screen shot of text

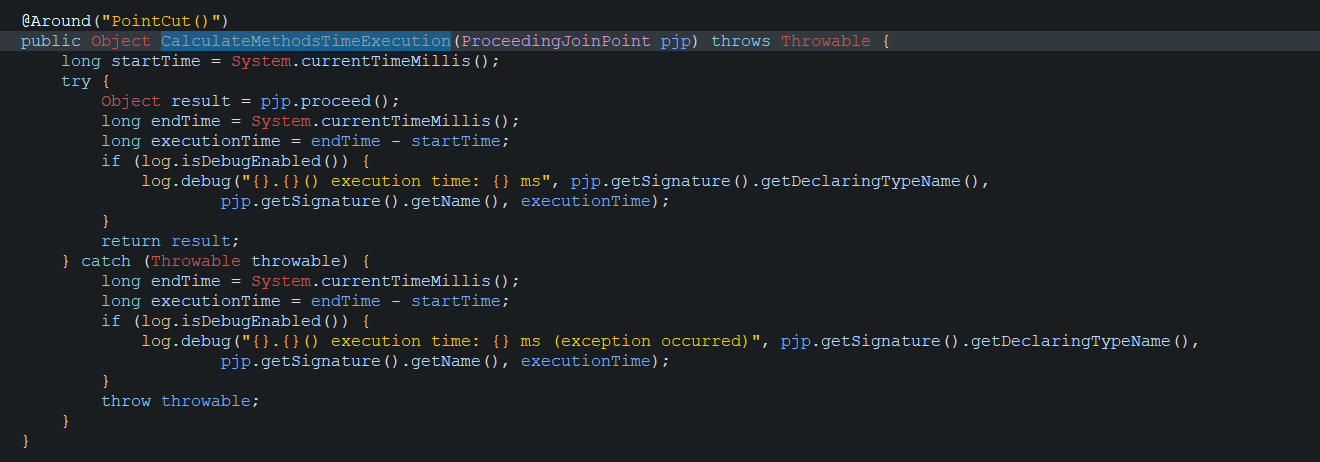
Description automatically generated

**Step 3:**  Add an aspect class. We create a pointcut with @Pointcut annotation and two advices with @Around

A computer code on a black background

Description automatically generated

Let’s write a new aspect. We’ll use Calculate methods time execution class from Spring Framework utils and @Around advice.



## **Results**

## API Test Results using Postman.

## **Step 1:** Send the POST API

A screenshot of a computer

Description automatically generated**Step 2:** Verify the LOGs

A black screen with white text

Description automatically generated

As we see log are shown in console by AOP implementation.

## **Conclusion**

In the realm of modern software development, robust logging is essential for maintaining application health. Through this documentation on \*\*Logging with Spring AOP\*\*, we've explored how to seamlessly integrate logging into Spring applications using Aspect-Oriented Programming (AOP).

By combining AOP principles and Spring's capabilities, we've uncovered a way to enhance traceability, debugging, and overall system reliability. Our journey covered understanding AOP's basics, integrating logging libraries, crafting customized log outputs, and strategic routing.

In closing, this synergy between Spring AOP and logging isn't just technical – it's strategic. Whether you're starting or fine-tuning your logging strategies, this guide equips you with insights for efficient troubleshooting and proactive maintenance. As you progress in software development, may these principles illuminate your path toward resilient Spring applications.