Pengenalan Aspect Oriented Programming

Pengenalan AOP

- AOP atau singkatan dari Aspect Oriented Programming
- AOP melengkapi OOP (Object Oriented Programming) dalam membuat kode program
- Saat kita menggunakan OOP, inti dari modularity kode adalah class, sedangkan di AOP, inti dari modularity kode adalah aspect
- Aspect memungkinkan modularity dari concerns (perhatian), yang bisa melintasi berbagai jenis tipe data dan object

Spring AOP

- Spring memiliki dua fitur AOP, yang pertama implementasi sendiri menggunaka Spring AOP, yang kedua menggunakan library AOP yang bernama AspectJ
- Pada kelas ini, kita akan membahas yang menggunakan library AspectJ, karena banyak digunakan di Spring, salah satunya untuk fitur Declarative Transaction Management

Membingungkan

- AOP biasanya akan sangat membingungkan ketika pertama kali mengenal istilahnya
- Tidak perlu khawatir, karena itu memang sudah biasa untuk programmer yang biasa menggunakan OOP
- Namun nanti ketika kita sudah lihat contoh-contoh penggunaan AOP, maka kita bisa dengan mudah mengerti tentang AOP

Pengenalan AspectJ

AspectJ

- AspectJ adalah salah satu library yang banyak digunakan untuk implementasi Aspect Oriented Programming
- Untungnya, Spring sudah terintegrasi dengan baik dengan AspectJ, sehingga kita tidak perlu banyak melakukan pengaturan AspectJ secara manual
- https://www.eclipse.org/aspectj/

Apakah Harus Belajar AspectJ Dulu?

- Sebelum belajar Spring AOP, apakah kita perlu belajar Library AspectJ terlebih dahulu?
- Tidak, di kelas ini saya akan bahas dari awal sehingga kita tidak perlu belajar Library AspectJ terlebih dahulu
- Namun jika tertarik untuk mendalami tentang AspectJ, bisa belajar di halaman dokumentasi resminya :
- https://www.eclipse.org/aspectj/doc/released/progguide/index.html

Membuat Project

Membuat Project

- https://start.spring.io/

Menambah Dependency

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

Tanpa AOP

Tanpa AOP

- Sekarang kita akan membuat contoh terlebih dahulu sebuah kode program tanpa AOP
- Kita akan membuat Service Class, lalu di tiap method di Service Class tersebut, kita akan selalu menambahkan log di awal method

Kode : Hello Service

```java
@Slf4j
@Component
public class HelloService {

    no usages  new *
    public String hello(String name) {
        log.info("Call HelloService.hello()");
        return "Hello " + name;
    }

    no usages  new *
    public String bye(String name) {
        log.info("Call HelloService.bye()");
        return "Bye " + name;
    }
```

Kode : Hello Service Unit Test

```java
@SpringBootTest
class HelloServiceTest {

    @Autowired
    private HelloService helloService;

    new *
    @Test
    void helloService() {
        Assertions.assertEquals("Hello Eko", helloService.hello("Eko"));
        Assertions.assertEquals("Bye Eko", helloService.bye("Eko"));
    }
}
```

Bayangkan

- Jika method terus bertambah, apa yang dilakukan?
- Maka kita harus membuat log terus menerus secara manual di tiap method nya
- Padahal jika diperhatikan, isi dari log polanya hampir sama, hanya menampilkan method apa yang sedang diakses
- AOP bisa membantu mempermudah ini, dimana kita bisa membuat Aspect yang melintasi semua method dari object tersebut, dimana di Aspect nya, kita hanya perlu menulis kode untuk log satu kali

Mengaktifkan AOP

Mengaktifkan AOP

- Secara default fitur AOP tidak berjalan di Spring
- Kita harus mengaktifkan secara manual menggunakan annotation EnableAspectJAutoProxy
- Kita bisa tambahkan di Bean Configuration agar fitur AOP aktif
- https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/EnableAspectJAutoProxy.html

Kode : Mengaktifkan AOP

```
@EnableAspectJAutoProxy
@SpringBootApplication
public class BelajarSpringAopApplication {

    new *
    public static void main(String[] args) {
        SpringApplication.run(BelajarSpringAopApplication.class, args);
    }

}
```

Aspect

Aspect

- Seperti yang dibahas sebelumnya, inti dari AOP adalah Aspect
- Oleh karena itu, hal yang harus kita lakukan adalah membuat Aspect
- Untuk membuat Aspect, kita cukup membuat Bean dan menambahkan annotation Aspect pada Bean tersebut
- Secara otomatis Spring akan membuatkan object Aspect dari class tersebut
- https://javadoc.io/doc/org.aspectj/aspectjrt/latest/org/aspectj/lang/annotation/Aspect.html

Kode : Log Aspect

```
no usages  new
@Aspect
@Component
public class LogAspect {
```

Kegunaan Aspect

- Secara default, Aspect tidak berguna jika kita tidak menambahkan behavior pada Aspect tersebut
- Oleh karena itu, kita perlu menambahkan behavior ke Aspect dengan cara menambahkan method pada Aspect tersebut
- Namun tidak sembarang method, ada ketentuannya

Join Point

Join Point

- Join Point adalah titik lokasi eksekusi program
- AspectJ sendiri sebenarnya mendukung banyak sekali Join Point, namun Spring AOP hanya mendukung Join Point pada eksekusi method di Bean

Contoh Join Point di Spring AOP

- Eksekusi method hello() di class HelloService
- Eksekusi semua method public di class HelloService
- Eksekusi semua method yang terdapat annotation @Test
- Eksekusi method di package service yang throw Exception
- Dan lain-lain
- Intinya adalah titik lokasi eksekusi method dengan kriteria tertentu, sehingga bisa melintasi satu atau lebih method dan object

Pointcut

Pointcut

- Pointcut adalah predikat yang cocok dengan Join Point
- Secara sederhana, Pointcut merupakan kondisi yang digunakan untuk menentukan Join Point
- Dan ketika kondisi terpenuhi, maka Aspect akan mengeksekusi Advice (akan dibahas di materi sendiri)
- Untuk membuat Pointcut, kita perlu menggunakan annotation Pointcut
- https://javadoc.io/doc/org.aspectj/aspectjrt/latest/org/aspectj/lang/annotation/Pointcut.html

Kode : Pointcut

```
@Aspect
public class LogAspect {

    no usages  new *
    @Pointcut("TODO change expression")
    public void helloServiceMethod() {
    }

}
    You, Moments ago • Uncommitted changes
```

Pointcut Expression

Pointcut Expression

- Saat kita membuat Pointcut, maka kita harus menambahkan expression yang berisi kondisi untuk Join Point nya
- Misal kita ingin membuat Pointcut untuk semua method di class HelloService, maka kita harus buat kondisi tersebut dalam bentuk Pointcut Expression
- AspectJ sebenarnya mendukung banyak sekali Pointcut Expression, namun Spring AOP hanya mendukung yang berhubungan dengan eksekusi method

Daftar Pointcut Expression (1)

Expression	Keterangan
execution	eksekusi method
within	object sesuai yang ditentukan
this	bean reference adalah instance tipe yang ditentukan
target	object adalah instance dari tipe yang ditentukan
args	argument method adalah instance dari tipe yang ditentukan

Daftar Pointcut Expression (2)

Expression	Keterangan
@target	object memiliki annotation yang ditentukan
@args	arguments method memiliki annotation yang ditentukan
@within	method di object yang memiliki annotation yang ditentukan
@annotation	method memiliki annotation yang ditentukan
bean	object dengan nama bean sesuai yang ditentukan

Kode : Pointcut Expression

```
no usages  new *
@Aspect
@Component
public class LogAspect {

    1 usage  new *
    @Pointcut("target(programmerzamannow.aop.service.HelloService)")
    public void helloServiceMethod() {
    }
```

Advice

Advice

- Advice adalah aksi yang yang dilakukan oleh Aspect pada Join Point
- Terdapat banyak sekali jenis Advice, misal Before, After, dan Around

Jenis Advice

Advice	Keterangan
Before	Aspect akan menjalankan aksi sebelum Join Point
AfterReturning	Aspect akan menjalankan aksi setelah Join Point return secara normal
AfterThrowing	Aspect akan menjalankan aksi setelah Joint Point throw exception
After	Aspect akan menjalankan aksi setelah selesai Join Point, After harus bisa menangani return normal atau throw exception
Around	Aspect memiliki kesempatan untuk menjalankan aksi sebelum dan setelah

Advices Annotation

- Semua jenis Advice direpresentasikan dalam annotation di package org.aspectj.lang.annotation
- https://javadoc.io/doc/org.aspectj/aspectjrt/latest/org/aspectj/lang/annotation/package-summary.html
- Saat menggunakan Advice, kita harus tentukan Pointcut yang akan kita gunakan, caranya dengan menyebutkan nama method dari Pointcut nya

Kode : Before

```java
@Aspect
@Component
@Slf4j
public class LogAspect {

    1 usage  new *
    @Pointcut("target(programmerzamannow.aop.service.HelloService)")
    public void helloServiceMethod() {
    }

    new *
    @Before("helloServiceMethod()")
    public void beforeHelloServiceMethod() {
        log.info("Before HelloService Method");
    }
```

Advice Parameter

Advice Parameter

- Saat kita membuat Advice, kita juga bisa mendapat informasi dari detail eksekusi method nya dari object JoinPoint
- Kita bisa tambahkan parameter JointPoint di method Advice yang kita buat
- https://javadoc.io/doc/org.aspectj/aspectjrt/latest/org/aspectj/lang/JoinPoint.html

Kode : Advice Parameter

```java
@Aspect
@Component
@Slf4j
public class LogAspect {

    1 usage  new *
    @Pointcut("target(programmerzamannow.aop.service.HelloService)")
    public void helloServiceMethod() {
    }

    new *
    @Before("helloServiceMethod()")
    public void beforeHelloServiceMethod(JoinPoint joinPoint) {
        String className = joinPoint.getTarget().getClass().getName();
        String methodName = joinPoint.getSignature().getName();
        log.info("Before " + className + "." + methodName + "()");
    }
```

Proceeding Join Point

Proceeding Join Point

- Khusus untuk Advice dengan jenis Around, maka kita gunakan parameter ProceedingJoinPoint, hal ini karena untuk Around, kita bisa melakukan sebelum dan setelah
- Dimana untuk mengeksekusi method aslinya dari Join Point, kita harus memanggil method ProceedingJoinPoint.proceed(args)
- https://javadoc.io/doc/org.aspectj/aspectjrt/latest/org/aspectj/lang/ProceedingJoinPoint.html

Kode : Advice Around

```
new *
@Around("helloServiceMethod()")
public Object aroundHelloServiceMethod(ProceedingJoinPoint joinPoint) throws Throwable {
    String className = joinPoint.getTarget().getClass().getName();
    String methodName = joinPoint.getSignature().getName();
    try {
        log.info("Around Before " + className + "." + methodName + "()");
        return joinPoint.proceed(joinPoint.getArgs());
    } catch (Throwable throwable) {
        log.info("Around Error " + className + "." + methodName + "()");
        throw throwable;
    } finally {
        log.info("Around Finally " + className + "." + methodName + "()");
    }
}
```

Pointcut Expression Format

Pointcut Expression Format

- Sebelumnya kita sudah coba menggunakan Pointcut Expression untuk target
- Tiap jenis Pointcut memiliki format sendiri-sendiri, dan yang paling sering digunakan biasanya adalah execution

Non Execution

- Untuk format yang Pointcut Expression yang buka execution sangat sederhana, cukup sebutkan nama target nya
- Target juga bisa menggunakan regex jika kita mau ke lebih dari satu Type

Contoh Non Execution

- within(com.xyz.service.*) : Semua method di bean di package service
- within(com.xyz.service..*) : Semua method di bean di package service dan di sub package nya
- target(com.xyz.service.AccountService) : Semua method di bean AccountService
- args(java.lang.String, java.lang.String) : Semua method di bean yang memiliki dua parameter String
- @target(org.springframework.transaction.annotation.Transactional) : Semua method yang memiliki annotation Transactional
- bean(traceService) : Semua method di bean dengan nama traceService
- bean(*Service) : Semua method di bean dengan akhiran Service

Execution

- Untuk execution, format expression nya lebih kompleks, yaitu dengan format :
- execution(modifier-pattern type-patter.method-pattern(param-pattern) throws-pattern)
- https://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html

Contoh Execution

- execution(public * *(..)) : Semua method public
- execution(* set*(..)) : Semua method dengan prefix set
- execution(* com.xyz.service.AccountService.*(..)) : Semua method di class AccountService
- execution(* com.xyz.service.*.*(..)) : Semua method di package service
- execution(* com.xyz.service..*.*(..)) : Semua method di package service dan sub package nya

Kode : Execution

```
1 usage  new *
@Pointcut("execution(* programmerzamannow.aop.service.HelloService.*(java.lang.String))")
public void pointcutHelloServiceStringParam() {
}

new *
@Before("pointcutHelloServiceStringParam()")
public void logStringParameter(JoinPoint joinPoint) {
    String value = (String) joinPoint.getArgs()[0];
    log.info("Execute method with parameter : " + value);
}
```

Multiple Pointcut

Multiple Pointcut

- Saat kita menggunakan Pointcut atau Advice, kita bisa menggunakan lebih dari satu Pointcut Expression
- Hal ini kadang bermanfaat ketika kita mau melakukan kombinasi untuk beberapa Pointcut sehingga lebih mudah digunakan
- Misal kita membuat pointcut khusus untuk package service
- Lalu kita membuat pointcut khusus untuk bean dengan suffix Service
- Lalu kita membuat pointcut khusus untuk public method
- Kita bisa gabung semuanya sehingga terbentuk pointcut baru, package service, semua bean yang suffix nya Service dan method nya public
- Kita bisa gunakan tanda && untuk menggabungkan Pointcut

Kode : Multiple Pointcut

```
1 usage  new *
@Pointcut("execution(* programmerzamannow.aop.service.*.*(..))")
public void pointcutServicePackage() {}

1 usage  new *
@Pointcut("bean(*Service)")
public void pointcutServiceBean() {}

1 usage  new *
@Pointcut("execution(public * *(..))")
public void pointcutPublicMethod() {}

no usages  new *
@Pointcut("pointcutServicePackage() && pointcutServiceBean() && pointcutPublicMethod()")
public void publicMethodForService() {}
```

Kode : Advice

```
1 usage  new *
@Pointcut("pointcutServicePackage() && pointcutServiceBean() && pointcutPublicMethod()")
public void publicMethodForService() {
}

new *
@Before("publicMethodForService()")
public void logAllServiceMethod() {
    log.info("Log for all service methods");
}
```

Passing Parameter

Passing Argument

- Saat kita membuat Advice, kadang kita ingin menangkap data parameter yang ada di Joint Point
- Sebenarnya kita bisa menggunakan object JointPoint.getArgs(), namun itu datanya berubah Object[]
- Kita bisa memanfaatkan Pointcut Expression args untuk mengambil data parameter, dan mengirimnya ke method Advice

Cara Mengirim Parameter

- Cukup gunakan args(value), artinya parameter akan dikirim ke parameter di advice dengan nama value
- args(value1, value2), artinya parameter akan dikirim ke parameter value1 dan value2 di advice
- Jika kita hanya peduli dengan parameter awal, kita bisa gunakan args(value1, value2,..)

Kode : Passing Parameter

```
1 usage  new *
@Pointcut("execution(* programmerzamannow.aop.service.HelloService.*(java.lang.String))")
public void pointcutHelloServiceStringParam() {
}

new *
@Before("pointcutHelloServiceStringParam() && args(name)")
public void logStringParameter(String name) {
    log.info("Execute method with parameter : " + name);
}
```