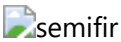
Spring AOP



Qu'est-ce que AOP ?

Aspect Oriented Programming

L'aspect orienté programme est un paradigme de programmation qui permet de séparer le code métier de la logique de gestion des erreurs, des logs, des transactions, etc.

Il consiste à créer des morceaux de code qui seront injectés dans le code métier sans avoir à modifier le code métier.

Exemple

Voici un exemple de code qui permet de générer un log à chaque fois qu'une méthode d'un controller est appelée.

```
@Aspect
@Component
public class LoggingAspect {

    private final Logger logger = LoggerFactory.getLogger(this.getClass());
```

```
    @Pointcut("within(@org.springframework.web.bind.annotation.RestController *)")
    public void controller() {
    }

    @Pointcut("execution(* *(..))")
    protected void allMethod() {
    }

    @Before("controller() && allMethod()")
    public void logBefore(JoinPoint joinPoint) {
        logger.info("Enter: {}.{}() with argument[s] = {}",
joinPoint.getSignature().getDeclaringTypeName(),
                joinPoint.getSignature().getName(),
Arrays.toString(joinPoint.getArgs()));
    }

    @AfterReturning(pointcut = "controller() && allMethod()", returning =
"result")
    public void logAfter(JoinPoint joinPoint, Object result) {
        logger.info("Exit: {}.{}() with result = {}",
joinPoint.getSignature().getDeclaringTypeName(),
                joinPoint.getSignature().getName(), result);
    }

    @AfterThrowing(pointcut = "controller() && allMethod()", throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
        logger.error("Exception in {}.{}() with cause = {}",
joinPoint.getSignature().getDeclaringTypeName(),
                joinPoint.getSignature().getName(), e.getCause() != null ?
e.getCause() : "NULL");
    }
}
```

Terminologie

Il faut connaître quelques termes pour comprendre le fonctionnement d'AOP.

Aspect

Un `aspect` est une unité de code qui encapsule une fonctionnalité transversale, comme la gestion des transactions, la journalisation, la sécurité, etc.

Les aspects peuvent être appliqués à des `points de coupe` spécifiques dans le code métier.

Point de coupe (Pointcut)

Un `point de coupe` est un endroit dans le code où un `aspect` peut être appliqué, comme les entrées et sorties de méthodes ou les levées d'exceptions.

Les `points de coupe` sont décrits par des `expressions de point de coupe`.

Expression de point de coupe (Pointcut Expression)

Une `expression de point de coupe` est une description de l'emplacement où un `aspect` doit être appliqué.

Les `expressions de point de coupe` peuvent être basées sur des annotations, des méthodes ou des types de données.

Advice (Greffon)

Un `advice` est une méthode d'un `aspect` qui est exécutée lorsqu'un `point de coupe` est atteint.

Il existe plusieurs types d'`advice`, comme @Around, @Before et @After qui définissent le moment où la méthode d'un `aspect` est exécutée par rapport à une méthode du code métier.

Jointure (JoinPoint)

Une `jointure` est le point où un `aspect` est appliqué à un `point de coupe`.

Il existe plusieurs types de `jointures`, comme @Before et @After, qui définissent le moment où l'aspect est appliqué par rapport à la méthode du code métier.

Comment créer un Aspect ?

Spring AOP utilise des annotations pour définir les aspects.

Il n'est pas exactement le même que la librairie AspectJ de Java.

@Aspect

Pour créer un aspect, il faut créer une classe avec l'annotation `@Aspect` et `@Component` ou `@Configuration`.

> La classe doit être un bean du contexte Spring.

```
@Aspect
@Component
public class LoggingAspect {
    ...
}
```

Les points de coupe

Les points de coupe sont décrits par des expressions de point de coupe.

Il existe plusieurs types d'expressions de point de coupe, comme `within`, `execution`, `this`, `target`, `args`, `@annotation`, etc.

---

within

L'expression de point de coupe `within` permet de définir un point de coupe sur les classes ou les interfaces.

Ici l'expression `within(com.example.demo.controller.*)` permet de définir un point de coupe sur toutes les classes du package `com.example.demo.controller`.

```
@Pointcut("within(com.example.demo.controller.*)")
public void controller() {
}
```

---

execution

L'expression de point de coupe `execution` permet de définir un point de coupe sur les méthodes.

Ici l'expression `execution(* *(..))` permet de définir un point de coupe sur toutes les méthodes.

```
@Pointcut("execution(* *(..))")
protected void allMethod() {
}
```

---

expression précise

Il est possible de cibler une expression plus précise avec le chemin complet de la classe et la méthode.

Le format de la chaîne est `execution(<Type de retour> <packages>.<Class>.<methodes>(<ParamsType>))`.

---

expression

Par exemple `execution(* com.exemple.demo.*Controller.delete*(..))` permet de définir un point de coupe sur toutes les méthodes commencant par `delete` des classes du package `com.exemple.demo` qui finissent par `Controller`.

```
@Before("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logBefore(JoinPoint joinPoint) {
    ...
}
```

---

this

L'expression de point de coupe `this` permet de définir un point de coupe sur les classes.

Ici l'expression `this(com.example.demo.controller.UserController)` permet de définir un point de coupe sur la classe `com.example.demo.controller.UserController`.

```
@Pointcut("this(com.example.demo.controller.UserController)")
public void thisUserController() {
}
```

---

target

L'expression de point de coupe `target` permet de définir un point de coupe sur les interfaces.

Ici l'expression `target(com.example.demo.services.UserService)` permet de définir un point de coupe sur l'interface `com.example.demo.services.UserService`.

```
@Pointcut("target(com.example.demo.services.UserService)")
public void targetUserController() {
}
```

---

args

L'expression de point de coupe `args` permet de définir un point de coupe sur les paramètres.

Ici l'expression `args(java.lang.String)` permet de définir un point de coupe sur les paramètres de type `java.lang.String`.

```
@Pointcut("args(java.lang.String)")
public void argsUserController() {
}
```

@annotation

L'expression de point de coupe `@annotation` permet de définir un point de coupe sur les annotations.

Ici l'expression `@annotation(com.example.demo.annotation.Loggable)` permet de définir un point de coupe sur les annotations `com.example.demo.annotation.Loggable`.

```
@Pointcut("@annotation(com.example.demo.annotation.Loggable)")
public void annotationUserController() {
}
```

Comment créer un Advice ?

Il existe plusieurs types d'`advice`, comme @Around, @Before et @After qui définissent la moment où la méthode d'un `aspect` est exécutée par rapport à une méthode du code métier.

@Before

L'`advice` `@Before` est exécuté avant la méthode du code métier.

Dans l'exemple ci-dessous, l'`advice` `logBefore` est exécuté avant les méthodes `delete` des controllers du package `com.exemple.demo` ne soient exécutées.

```
@Before("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logBefore(JoinPoint joinPoint) {
    ...
}
```

@After

L'`advice` `@After` est exécuté après la méthode du code métier.

Dans l'exemple ci-dessous, l'`advice` `logAfter` est exécuté après les méthodes `delete` des controllers du package `com.exemple.demo` ne soient exécutées.

```
@After("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logAfter(JoinPoint joinPoint) {
    ...
}
```

@AfterReturning

L'`advice` `@AfterReturning` est exécuté après la méthode du code métier et si elle retourne une valeur.

Dans l'exemple ci-dessous, l'`advice` `logAfterReturning` est exécuté après les méthodes `delete` des controllers du package `com.exemple.demo` ne soient exécutées et si elles retournent une valeur.

```
@AfterReturning("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logAfterReturning(JoinPoint joinPoint) {
    ...
}
```

@AfterThrowing

L'`advice @AfterThrowing` est exécuté après la méthode du code métier et si elle retourne une exception.

Dans l'exemple ci-dessous, l'`advice logAfterThrowing` est exécuté après les méthodes `delete` des controllers du package `com.exemple.demo` ne soient exécutées et si elles retournent une exception.

```
@AfterThrowing("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logAfterThrowing(JoinPoint joinPoint) {
    ...
}
```

JoinPoint

Les méthodes des `advice` peuvent prendre en paramètre un objet `JoinPoint` qui permet d'accéder à la méthode du code métier.

```
@Before("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logBefore(JoinPoint joinPoint) {
    String methodName = joinPoint.getSignature().getName();
    Object[] args = joinPoint.getArgs();
    ...
}
```

@Around

L'`advice @Around` est exécuté avant et après la méthode du code métier.

Dans l'exemple ci-dessous, l'`advice logAround` est exécuté avant et après les méthodes `delete` des controllers du package `com.exemple.demo` ne soient exécutées.

```
@Around("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logAround(ProceedingJoinPoint joinPoint) {
    // code avant la méthode du code métier
    joinPoint.proceed(); // exécute la méthode du code métier
    // code après la méthode du code métier
}
```

ProceedingJoinPoint

L'objet `ProceedingJoinPoint` permet d'exécuter la méthode du code métier.

```
@Around("execution(* com.exemple.demo.*Controller.delete*(..))")
public void logAround(ProceedingJoinPoint joinPoint) {
    // code avant la méthode du code métier
    joinPoint.proceed(); // exécute la méthode du code métier
    // code après la méthode du code métier
}
```

Attention!

Vous pouvez empêcher l'exécution de la méthode du code métier en ne faisant pas appel à la méthode `proceed()`.

Il devient compliqué pour une autre personne que comprendre le code et de savoir pourquoi la méthode du code métier n'est pas exécutée.

Exemple de cas d'utilisation

Pour les logs

Vous pouvez utiliser un `aspect` pour logger les méthodes du code métier.

```
@Aspect
@Component
public class LogAspect {

    private static final Logger logger = LoggerFactory.getLogger(LogAspect.class);

    @Around("execution(* com.exemple.demo.*Controller.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.currentTimeMillis();
        Object result = joinPoint.proceed();
        long executionTime = System.currentTimeMillis() - start;

        logger.info(joinPoint.getSignature() + " executed in " + executionTime +
"ms");
        return result;
    }
}
```

Pour les transactions

Vous pouvez utiliser un `aspect` pour gérer les transactions.

```
@Aspect
@Component
public class TransactionAspect {

    @Around("execution(* com.exemple.demo.*Controller.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        TransactionStatus status = transactionManager.getTransaction(new
DefaultTransactionDefinition());
        try {
            Object result = joinPoint.proceed();
            transactionManager.commit(status);
            return result;
        } catch (Throwable ex) {
            transactionManager.rollback(status);
            throw ex;
        }
    }
}
```

Pour les autorisations

Vous pouvez utiliser un `aspect` pour gérer les autorisations.

```
@Aspect
@Component
public class AuthorizationAspect {

    @Around("execution(* com.exemple.demo.*Controller.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        if (authentication == null || !authentication.isAuthenticated()) {
            throw new UnauthorizedException();
        }
        return joinPoint.proceed();
    }
}
```

Exercice 8