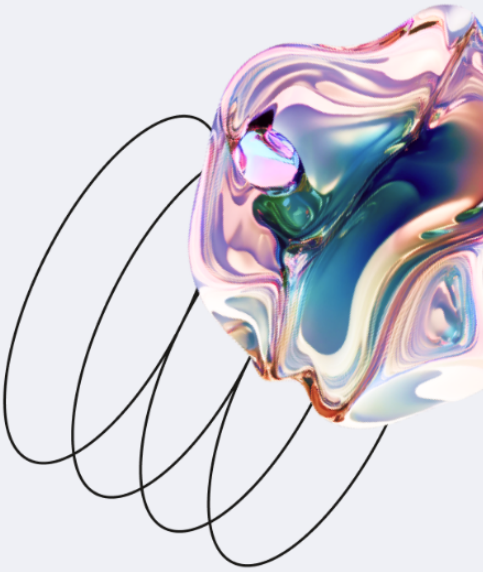
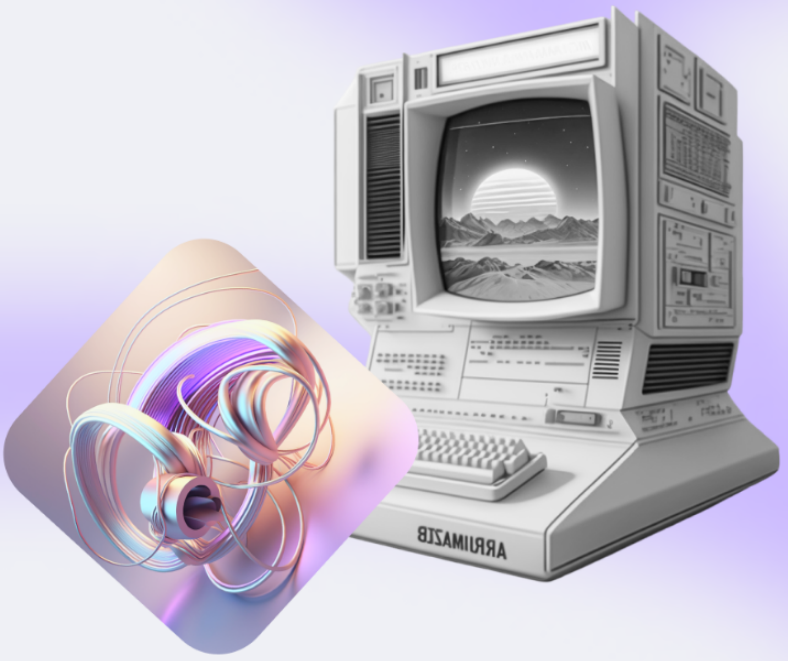
GeekBrains



Spring AOP. Управление транзакциями

Фреймворк Spring



Оглавление

Введение

Привет, друзья! 🌟 Сегодня у нас особенный день, потому что мы окунемся в мир Spring AOP. Но перед тем как погрузиться в глубины, давайте немного освежим в памяти, что мы уже изучали и почему это так важно.

Вы помните наши занятия по системам сборки Maven и Gradle, правда? Как можно забыть те моменты, когда мы с вами боролись с зависимостями, старались сделать

наш проект максимально автоматизированным и понимали всю важность правильной структурированности? Это было интересно и вызывающе!

И конечно, как забыть Spring MVC! Мы учились создавать красивые, отзывчивые и функциональные веб-приложения, манипулировали данными и сталкивались с множеством интересных задач.

Следующий этап нашего путешествия был посвящен Spring Security. Здесь мы обсуждали вопросы безопасности, авторизации, аутентификации и поняли, как важно уметь защитить свои приложения от злонамеренных действий.

Spring Data стал нашим спутником в понимании того, как можно легко и удобно работать с базами данных, делая это эффективно и без лишних сложностей.

И, наконец, мы рассмотрели проектирование API, поняв, как важно уметь создавать качественные, понятные и стабильные API для наших клиентов.

Каждый из этих этапов дал нам инструменты и знания, которые делают нас профессионалами в мире разработки на Spring. И вот, объединив все это воедино, мы приблизились к новой теме, которую мы сегодня начнем разбирать – Spring AOP.

Что это такое и для чего оно нужно? Как это может помочь нам в нашей разработке? Все ответы на эти вопросы мы найдем в следующих разделах. Погрузимся в это вместе и откроем для себя новые горизонты!

Термины, используемые в лекции

Aspect – модуль, который определяет Advices, срезы и привязки. Это кросс-функциональные заботы, такие как логирование, аудит, безопасность и т. д.

Join Point – точка в программе, такая как выполнение метода, где можно применить Advices (advice). В Spring AOP, join points представляют собой выполнение методов.

Advice – действие, предпринимаемое аспектом в определенной точке соединения. Существуют разные типы Advices, такие как before, after, after-returning, after-throwing, и around.

Pointcut – выражение, которое выбирает определенные join points. Advices применяются к join points, выбранным через pointcuts.

Target – объект, к которому применяется Advices.

Proxy – объект, созданный после применения Advicesa к целевому объекту.

Weaving – процесс комбинирования аспектов с другим типом приложения для создания прокси-объекта. Это может быть выполнено во время компиляции (CTW), загрузки класса (LTW) или во время выполнения.

Introduction (Inter-type declaration) – добавление новых методов или свойств в существующие классы.

Транзакция – последовательность действий, которые либо полностью выполняются, либо полностью отменяются.

ACID – принципы транзакций — Атомарность, Согласованность, Изоляция и Долговечность (Atomicity, Consistency, Isolation, Durability).

Propagation – определяет, как транзакции относятся к друг другу. Например, REQUIRED, REQUIRES_NEW, SUPPORTS и т. д.

Isolation – уровень изоляции транзакции определяет, как данные, доступные одной транзакции, становятся видимыми для других.

@Transactional – аннотация Spring для объявления транзакционного метода.

Transaction Manager – компонент, который управляет транзакциями. Например, DataSourceTransactionManager для JDBC.

Rollback – отмена изменений, выполненных в рамках транзакции.

Spring AOP

Друзья, перед нами стоит задача разобраться, что же такое Spring AOP. Я обещаю, что не буду загружать вас сложными терминами, а попробую объяснить на простых и понятных примерах. Итак, начнем!

Первое, что нужно сказать: **AOP** это сокращение от **Aspect-Oriented Programming**. Так, знаю, звучит сложно. Но держитесь, я расскажу просто.

Представьте, что у вас есть дом. В этом доме есть разные комнаты: кухня, спальня, гостиная и так далее. Теперь представьте, что вы решаете установить в доме систему безопасности. В каждой комнате вы ставите датчик движения. Но ведь

датчик движения — это не основная часть комнаты, верно? Он просто добавляется для выполнения какой-то дополнительной функции.

В этом примере каждая комната — это какой-то отдельный модуль вашего приложения. А система безопасности, которую вы решили добавить — это пример функции, которую вы хотите “вплести” в разные части вашего приложения, не меняя их основного кода. Именно это и позволяет сделать AOP!

Теперь, давайте поговорим о разнице между **AOP** и **ООП**. **ООП**, или объектно-ориентированное программирование, — это способ организации кода, когда все строится вокруг объектов и их взаимодействий. Как будто у вас есть игрушечные фигурки (объекты), каждая из которых может что-то делать (методы) и иметь свои характеристики (атрибуты).

Тогда как **AOP** — это скорее “волшебный плащ”, который можно накинуть на несколько из этих фигурок одновременно, чтобы добавить им какую-то дополнительную функцию, не меняя их самих. Это не новый способ организации кода, а дополнение к существующим методикам, позволяющее легко внедрять функции в разные части приложения.

Надеюсь, теперь стало немного яснее. Мы только начали наше путешествие в мир Spring AOP, так что готовьтесь к новым интересным открытиям!

Взаимосвязь с предыдущими темами

Прежде чем говорить о важности Spring AOP, давайте разберемся, как этот компонент связан с теми, которые мы уже прошли.

Помните наш разговор о Spring MVC? Так вот, когда вы создаете контроллеры и обрабатываете запросы, иногда вы хотите добавить дополнительное поведение к этим контроллерам. Например, залогировать запросы или измерить время их выполнения. Вместо того чтобы добавлять эти дополнительные функции в каждый контроллер, вы можете использовать Spring AOP, чтобы автоматически применять это поведение ко многим частям вашего приложения.

А что насчет Spring Security? Часто нам нужно управлять доступом к определенным методам или функциям приложения. С помощью Spring AOP, вы можете легко добавить аспекты безопасности на нужные вам уровни приложения.

Теперь давайте представим, что у вас есть сложный проект, где вы используете Spring Data для работы с базой данных. И вам нужно залогировать все запросы к базе или измерить, сколько времени занимает каждый запрос. Опять-таки, вместо

того чтобы добавлять логику в каждый репозиторий или сервис, вы можете использовать Spring AOP для этой цели.

Таким образом, Spring AOP как бы вяжет воедино все наши предыдущие темы, добавляя дополнительную мощь и гибкость.

Важность в коммерческой разработке

Теперь, когда мы знаем, как Spring AOP связан с нашими предыдущими уроками, давайте поговорим о том, как это используется на практике, особенно в крупных компаниях.

В первую очередь, Spring AOP помогает разработчикам писать более чистый и модульный код. Вы можете сосредоточиться на основной логике вашего приложения, не беспокоясь о дополнительных функциях, таких как логирование или безопасность. Эти “перекрестные” задачи (cross-cutting concerns) легко управляются с помощью аспектов в AOP.

В крупных компаниях, где десятки, если не сотни разработчиков работают над сложными и масштабными проектами, управление такими “перекрестными” задачами может стать настоящим вызовом. И здесь Spring AOP может стать настоящим спасением.

Например, представьте, что в вашей компании решили изменить формат логирования. Если вы не используете AOP, вам, возможно, придется искать и менять код логирования в сотнях мест. Но с AOP вы просто обновляете один аспект, и изменения автоматически применяются во всем проекте.

В заключение, Spring AOP - это мощный инструмент, который помогает управлять повторяющимися и “перекрестными” задачами в вашем приложении. В крупных коммерческих проектах это может существенно ускорить разработку, упростить поддержку кода и улучшить качество продукта.

Аспекты в Spring AOP

Что такое аспект?

Вернемся к нашему примеру с домом, где мы установили датчики движения. Если бы этот дом был нашим приложением, а датчики - определенной функциональностью, которую мы хотим добавить, то сама система безопасности (вся схема с датчиками, сигнализацией и так далее) была бы аспектом.

Аспект в Spring AOP — это модуль, который определяет “перекрестные” или “сквозные” задачи, такие как логирование, безопасность или транзакции. Эти задачи обычно затрагивают многие части приложения и не связаны напрямую с бизнес-логикой.

Как аспекты работают?

Аспекты работают таким образом, что они “внедряются” или “вплетаются” в ваш код в определенные места, которые вы указываете. Это похоже на то, как вы можете настроить датчики движения в разных комнатах вашего дома. Когда происходит какое-то событие (например, движение в комнате), датчик реагирует на него. Аналогично, когда определенная часть вашего кода выполняется, аспект может “реагировать” на это, выполняя нужные действия.

Зачем это нужно?

Представьте, что у вас есть огромный пазл, где каждая деталька — это часть вашего приложения. Аспекты позволяют нам не изменять каждую детальку отдельно, а “налепить” на несколько из них стикер с дополнительной информацией или функциональностью. Это экономит время, уменьшает ошибки и делает код более чистым и организованным.

В больших проектах аспекты становятся незаменимыми, потому что представьте, что при изменении требований вам нужно менять одну и ту же функцию в десятках или даже сотнях мест. С аспектами вы просто меняете функциональность один раз, и она автоматически применяется везде, где нужно.

Аспекты в Spring AOP — это способ организовать “сквозную” функциональность в вашем приложении без необходимости менять основной код. Это как магические стикеры, которые вы можете приклеить к любой части вашего кода, чтобы добавить дополнительное поведение или функциональность.

Примеры использования аспектов в Spring AOP

Давайте погрузимся в конкретные примеры, чтобы понять, как это работает на практике!

1. Логирование методов

Допустим, вы хотите логировать каждый вызов определенного метода. Вместо того чтобы добавлять логирующий код в каждый метод, вы можете создать аспект для этого.

```
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.service.*.*(..))")
    public void logBeforeMethodCall(JoinPoint joinPoint) {
        System.out.println("Метод " + joinPoint.getSignature().getName() + "
был вызван");
    }
}
```

В этом примере @Before говорит о том, что аспект должен выполняться перед вызовом метода, а выражение в execution() указывает на то, какие именно методы нужно “перехватить”. В данном случае перехватываются все методы из всех классов в пакете com.example.service.

2. Измерение времени выполнения

Если вы хотите измерять, сколько времени занимает выполнение определенного метода, аспект может пригодиться.

```
@Aspect
@Component
public class PerformanceAspect {

    @Around("execution(* com.example.service.*.*(..))")
        public Object measureMethodExecutionTime(ProceedingJoinPoint
joinPoint) throws Throwable {
        long start = System.currentTimeMillis();

        Object result = joinPoint.proceed();

        long elapsedTime = System.currentTimeMillis() - start;
        System.out.println("Метод " + joinPoint.getSignature().getName() + "
выполнился за " + elapsedTime + " миллисекунд");
```

```
        return result;
    }
}
```

Здесь используется аннотация @Around, которая позволяет аспекту выполниться до и после метода, а также вокруг него.

3. Изменение возвращаемого значения

Аспекты также могут изменять возвращаемое значение метода.

```
@Aspect
@Component
public class ChangeReturnValueAspect {

    @AfterReturning(pointcut = "execution(* com.example.service.getName(..))",
returning = "result")
    public void changeName(JoinPoint joinPoint, String result) {
        result = "Измененное имя";
    }
}
```

В этом примере после вызова метода getName из пакета com.example.service, его возвращаемое значение будет изменено на “Измененное имя”.

Эти примеры показывают базовые возможности Spring AOP и то, как вы можете использовать аспекты для добавления дополнительной функциональности в ваш код без изменения самого кода.

Компоненты аспектов

После того, как мы познакомились с основной идеей аспектов, настало время погрузиться поглубже и понять, какие именно инструменты предоставляет нам Spring AOP для работы с аспектами. Основные из них — это Advices, Introductions и Around-Advices. Подсветим каждую из этих компонент.

1. Advices

Advice – это действие, которое выполняется аспектом в определенное время (например, перед или после выполнения метода). В Spring AOP существует несколько типов Advices, включая:

- **Before Advice**: Выполняется до целевого метода. Примером может служить логирование или проверка прав доступа.
- **After Returning Advice**: Выполняется после того, как целевой метод успешно завершил свою работу (без исключений). Можно использовать для дополнительной обработки результата метода.
- **After Throwing Advice**: Выполняется, если во время выполнения метода возникло исключение.
- **After (or After Finally) Advice**: Выполняется в любом случае после вызова метода, независимо от того, было ли исключение или нет.

Все эти advices позволяют нам вмешиваться в жизненный цикл вызова метода на разных этапах.

2. Introductions

Introduction (или “Mixin”) позволяет добавлять новые методы или свойства в существующие бины. Другими словами, с помощью introductions вы можете добавить новую функциональность к уже существующим объектам в вашем приложении без изменения их кода.

Это немного похоже на то, как если бы у вас был старый телефон, и вдруг, благодаря волшебной палочке, он получил новую функцию — например, стал поддерживать беспроводное зарядное устройство, хотя до этого такой функции у него не было.

3. Around-Advices

Around-Advice — это, пожалуй, самый мощный тип advice. Он объединяет в себе все остальные типы advices, так как позволяет вам вмешиваться в вызов метода до его выполнения, после него и даже изменять возвращаемое значение или кидать исключение вместо целевого метода.

Вернемся к нашему примеру с домом: это как если бы у вас была магическая дверь, через которую все гости должны проходить. Вы можете настроить эту дверь таким образом, чтобы она играла музыку, когда кто-то заходит, давала гостям

возможность оставить послание или даже блокировала вход для определенных людей.

Around-Advice предоставляет максимальный контроль над тем, как метод будет выполняться, что делает его очень мощным инструментом в арсенале Spring AOP.

Теперь, осознав различные компоненты аспектов, вы можете лучше понимать, как и когда использовать каждую из этих функций для достижения максимальной эффективности вашего приложения на Spring.

Углубляемся в Advices

Advices в Spring AOP играют ключевую роль в определении поведения аспектов. Они описывают, что именно делает аспект в определенные моменты выполнения программы. Иными словами, если бы ваш аспект был сценаристом фильма, то advices определили бы, какие именно сцены и когда следует показать.

Давайте рассмотрим каждый из видов advices подробнее:

1. Before Advice

Как можно догадаться по названию, “Before Advice” выполняется до вызова целевого метода. Но что это значит на практике? Это как будто перед тем, как вы начнете готовить обед, зазвонил будильник, напоминающий о том, что нужно помыть руки. “Before Advice” действует как этот будильник — он перехватывает выполнение и добавляет действие, которое должно произойти до основного действия.

2. After Returning Advice

Этот тип advice выполняется после того, как целевой метод успешно завершен, и возвращает результат. Представьте, что после приготовления обеда у вас есть традиция записывать, какой ингредиент вы использовали больше всего. “After Returning Advice” именно так и работает: он отслеживает результат и позволяет вам добавить дополнительное действие после успешного завершения основного метода.

3. After Throwing Advice

Если в процессе выполнения целевого метода возникает исключение, именно здесь вступает в действие “After Throwing Advice”. Это как если бы вы пытались приготовить новое блюдо, и что-то пошло не так — например, вы пересолили. Вместо того чтобы просто отбросить все и начать сначала, вы решили бы заметить свою ошибку и занести её в книгу кулинарных ошибок. Этот advice помогает обрабатывать исключения и реагировать на них соответствующим образом.

4. After (or After Finally) Advice

“After Advice” гарантирует, что определенный код будет выполнен после вызова целевого метода, независимо от того, завершился ли метод успешно или с исключением. Это как если бы после приготовления обеда, независимо от результата, вы бы всегда молились в благодарность за еду. Это гарантированный шаг, который следует сразу после основного действия.

5. Around Advice

Этот тип advice был упомянут ранее и является одним из самых мощных. Он позволяет вам обернуть вызов метода, добавив код до и после его выполнения, а также модифицировать результат или исключение. Это как если бы вы могли контролировать весь процесс приготовления обеда от начала до конца, включая выбор ингредиентов, способ приготовления и даже презентацию блюда на столе.

Понимание различных advices и их возможностей критически важно для эффективного использования AOP в Spring, так как именно они определяют, как и когда ваш аспект будет вмешиваться в выполнение приложения.

Before Advice

Before Advice — это действительно интересная штука. Если рассматривать его в контексте кино, то это как предыстория перед основным фильмом или как короткая реклама перед основным шоу.

Что такое Before Advice?

“Before Advice” в Spring AOP позволяет вам выполнять определенный код перед вызовом целевого метода. Вы можете считать его своеобразным “стражем” или “предвестником”, который выполняет свою работу прежде, чем основное действие начинается.

Зачем это может быть полезно?

В мире программирования есть множество ситуаций, когда перед выполнением основного метода необходимо выполнить предварительную проверку или какое-то действие. Например:

1. Проверить, имеет ли пользователь необходимые права доступа для вызова метода.
2. Залогировать информацию о вызове метода.
3. Инициировать какой-то ресурс.

Пример кода с Before Advice:

Допустим, у нас есть веб-приложение, и мы хотим логировать информацию о том, когда пользователи пытаются получить доступ к некоторой защищенной странице.

1. Сначала определим наш аспект:

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

2. Теперь добавим Before Advice, который будет логировать каждый вызов метода viewProtectedPage():

```
@Before("execution(* com.example.service.UserService.viewProtectedPage(..))")
public void logBeforeAccess() {
    System.out.println("Попытка доступа к защищенной странице!");
}
```

В этом коде execution(* com.example.service.UserService.viewProtectedPage(..)) — это точка соединения (pointcut), которая указывает, когда именно будет срабатывать наш advice. В нашем случае, он срабатывает перед каждым вызовом метода viewProtectedPage() в классе UserService.

Таким образом, каждый раз, когда этот метод вызывается, перед его выполнением в консоль будет выводиться сообщение “Попытка доступа к защищенной странице!”.

“Before Advice” действительно мощный инструмент в вашем арсенале Spring AOP, позволяя вам легко добавлять дополнительную логику перед основным поведением вашего приложения.

After Advice

Думаю, многие из нас помнят момент в кино, когда после основного фильма появляется дополнительная сцена. Это чаще всего дает нам какой-то интересный поворот или намек на будущие события. В мире Spring AOP, “After Advice” действует именно так!

Что такое After Advice?

“After Advice” в Spring AOP позволяет вам выполнить код непосредственно после вызова целевого метода, независимо от того, был ли вызов успешным или завершился исключением. Это похоже на тот момент, когда, закончив книгу, вы начинаете прочитывать послесловие автора.

Почему это может быть полезно?

1. Логирование завершения работы метода или результата его работы.
2. Освобождение ресурсов.
3. Закрытие соединений или других важных операций, которые должны быть выполнены после основного действия.

Пример кода с After Advice:

Представим, что у нас есть приложение, в котором мы хотим отслеживать, как часто пользователи завершают определенные задания.

1. Создадим аспект:

```
@Component
@Aspect
public class TaskCompletionAspect {
}
```

2. Добавим After Advice для отслеживания завершения задания:

```
@After("execution(* com.example.service.TaskService.completeTask(..))")
public void logAfterTaskCompletion() {
```

```
        System.out.println("Задание успешно завершено!");
}
```

Здесь execution(* com.example.service.TaskService.completeTask(..)) — это точка соединения (pointcut), которая указывает, когда будет активирован наш advice. В данном случае, после каждого вызова метода completeTask() в классе TaskService.

Так, после завершения задания в консоли мы увидим сообщение: “Задание успешно завершено!”.

“After Advice” отличный способ убедиться, что определенный код будет выполнен после основной логики в вашем приложении, что делает его незаменимым инструментом при создании надежных и масштабируемых приложений на Spring.

AfterReturning Advice

Помните, когда вы впервые приготовили что-то сложное, например, торт, и не могли дождаться, чтобы узнать, насколько вкусно получилось? Или когда вы делаете что-то важное и хотите быть уверены, что все прошло гладко? Вот именно этот момент охватывает “AfterReturning Advice” в мире Spring AOP!

Что такое AfterReturning Advice?

“AfterReturning Advice” в Spring AOP позволяет вам выполнить код после успешного завершения целевого метода, то есть когда метод выполнен и не выбрасывает исключения. Этот advice особенно полезен, когда вы хотите что-то сделать с возвращаемым значением метода.

Почему это может быть полезно?

1. Модифицировать возвращаемое значение.
2. Логировать результат работы метода.
3. Выполнить дополнительные проверки или действия на основе возвращаемого значения.

Пример кода с AfterReturning Advice:

Допустим, у нас есть сервис, который возвращает некоторое сообщение для пользователя на основе его действий в приложении.

1. Определите аспект:

```
@Component
@Aspect
public class MessageAspect {
}
```

2. Добавим AfterReturning Advice, чтобы логировать возвращаемое сообщение:

```
@AfterReturning(pointcut = "execution(* com.example.service.MessageService.getUserMessage(..))", returning = "message")
public void logUserMessage(String message) {
    System.out.println("Возвращаемое сообщение: " + message);
}
```

Здесь execution(* com.example.service.MessageService.getUserMessage(..)) — это точка соединения (pointcut), которая определяет, когда срабатывает наш advice. С помощью атрибута returning, мы указываем, что хотим получить возвращаемое значение метода и использовать его в нашем advice.

Таким образом, когда метод getUserMessage() будет вызван и успешно завершен, в консоль будет выводиться сообщение, возвращаемое этим методом.

“AfterReturning Advice” предоставляет нам мощный инструмент для работы с возвращаемыми значениями методов, что позволяет добавить дополнительную функциональность и контроль в наше приложение.

AfterThrowing Advice

Представьте, что вы катаетесь на горных лыжах. Всё идет отлично, но внезапно вы падаете. Было бы здорово, если бы рядом был друг, который помог бы вам подняться и удостоверился, что с вами всё в порядке, не так ли? В мире программирования “AfterThrowing Advice” играет роль этого заботливого друга, когда что-то идет не так.

Что такое AfterThrowing Advice?

“AfterThrowing Advice” в Spring AOP позволяет вам выполнить код в случае, если целевой метод выбрасывает исключение. Это отличный способ управлять исключениями и предоставлять информацию о том, что пошло не так.

Почему это может быть полезно?

1. Логирование исключений.
2. Отправка оповещений разработчикам или системам мониторинга.

3. Предоставление пользователю дружественного сообщения об ошибке.

Пример кода с AfterThrowing Advice:

Допустим, у нас есть сервис для обработки платежей, и мы хотим узнать, когда происходит ошибка.

1. Определите аспект:

```
@Component
@Aspect
public class PaymentAspect {
}
```

2. Добавим AfterThrowing Advice, чтобы логировать исключение:

```
@AfterThrowing(pointcut = "execution(* com.example.service.PaymentService.processPayment(..))", throwing = "exception")
public void logPaymentError(Exception exception) {
        System.out.println("Произошла ошибка при обработке платежа: " + exception.getMessage());
}
```

Здесь execution(* com.example.service.PaymentService.processPayment(..)) — это точка соединения (pointcut). Атрибут throwing позволяет нам захватить исключение, выброшенное целевым методом, и использовать его в нашем advice.

Таким образом, если метод processPayment() выбросит исключение, мы увидим сообщение об ошибке в консоли.

“AfterThrowing Advice” дает нам возможность эффективно реагировать на исключения и улучшать устойчивость нашего приложения к ошибкам.

Around Advice

Знаете те моменты, когда вам нужно подготовиться перед чем-то важным и сделать что-то после этого? Как если бы вы готовились к важной встрече: вы бы подготовились, провели саму встречу, а потом обдумали, как всё прошло. “Around Advice” — это именно такой “мульти-инструмент” в мире AOP. Он позволяет вам вмешиваться ДО и ПОСЛЕ выполнения целевого метода, а также, при необходимости, модифицировать его выполнение.

Что такое Around Advice?

“Around Advice” в Spring AOP оборачивает целевой метод, предоставляя вам возможность выполнения действий и до, и после его вызова, а также контролировать сам процесс вызова. Это самый мощный тип advice, так как дает полный контроль над выполнением метода.

Почему это может быть полезно?

1. Измерение времени выполнения метода.
2. Транзакционное управление.
3. Изменение аргументов или возвращаемого значения метода.
4. Предотвращение вызова целевого метода под определенными условиями.

Пример кода с Around Advice:

Представим, что вы хотите измерить, сколько времени занимает выполнение определенного метода в вашем сервисе.

1. Определите аспект:

```
@Component
@Aspect
public class PerformanceAspect {
}
```

2. Добавим Around Advice для измерения времени выполнения:

```
@Around("execution(* com.example.service.HeavyDutyService.performTask(..))")
public Object measureExecutionTime(ProceedingJoinPoint joinPoint)
throws Throwable {
    long startTime = System.currentTimeMillis();

    Object result = joinPoint.proceed();  // вызов целевого метода

    long endTime = System.currentTimeMillis();

     System.out.println("Метод " + joinPoint.getSignature() + " выполнялся " +
(endTime - startTime) + " миллисекунд.");
```

```
    return result;   // возвращаем результат выполнения целевого метода
}
```

Здесь метод joinPoint.proceed() вызывает целевой метод. Время, затраченное на его выполнение, измеряется и выводится в консоли.

“Around Advice” — это, безусловно, мощный инструмент в вашем арсенале Spring AOP, который предоставляет гибкость и контроль над выполнением методов.

Join Points

Давайте представим себе театральную постановку. У вас есть актёры, декорации, освещение и множество других элементов, которые работают вместе, чтобы создать потрясающее представление. Но иногда режиссёру нужно сделать корректировки: изменить момент входа актёра на сцену или добавить эффекты освещения в определенный момент. Эти моменты, когда режиссёру нужно вмешаться, можно сравнить с “Join Points” в мире Spring AOP.

Что такое Join Points?

Join Point в Spring AOP — это место в программе, где аспект может быть применен. Это может быть при вызове метода, при обработке исключения, при инициализации объекта и так далее. Join Points обозначают конкретные моменты в выполнении программы, когда можно “вмешаться” с помощью аспектов.

Каковы основные типы Join Points?

Несмотря на то что в теории может существовать множество различных Join Points, в Spring AOP мы обычно имеем дело с вызовами методов. Это означает, что наиболее распространенным типом Join Point является момент, когда метод вызывается.

Почему это важно?

Понимание Join Points критически важно для успешного применения AOP. Это как если бы режиссёр театральной постановки не знал, когда актёры входят на сцену или когда происходят ключевые моменты действия. Если вы не знаете, где находятся ваши Join Points, вы не сможете эффективно применять ваши аспекты.

Аналогия с книгой

Представьте, что ваше приложение — это книга. Каждая глава книги — это определенный функционал вашего приложения, а каждый абзац — это метод. Join Points в этой аналогии будут ключевыми словами или фразами, которые вы хотите выделить или комментировать. Когда вы найдете такое ключевое слово (Join Point), вы можете добавить примечание или комментарий (Advice) в этом месте.

Join Points — это основа AOP. Они указывают, где и когда ваши аспекты будут применяться, давая вам возможность модифицировать или улучшать функциональность вашего приложения.

Практика использования Join Points

Подойдём к этому делу практически! Вспомним, что Join Points — это конкретные точки в вашем коде, где аспект может “вмешаться”. Самый популярный тип Join Point в Spring AOP — это вызов метода. Давайте разберём это на примерах.

Определение Join Points

Для того чтобы определить, где именно ваш аспект должен “вступать в игру”, вы используете выражения Pointcut. Эти выражения позволяют вам выбирать, на какие методы, классы или даже пакеты должен реагировать ваш аспект.

Пример 1: Применение аспекта ко всем методам в классе

```
@Aspect
@Component
public class MyAspect {

    @Before("execution(* com.example.service.MyService.*(..))")
    public void beforeAnyMethodInMyService() {
        System.out.println("Вызван метод в MyService!");
    }
}
```

В данном примере execution(* com.example.service.MyService.*(..)) — это Pointcut выражение. Оно говорит Spring AOP применить аспект перед выполнением любого метода (.*(..)) в классе MyService.

Пример 2: Применение аспекта ко всем методам в пакете

```
@Aspect
@Component
public class MyAspect {

    @Before("execution(* com.example.service.*.*(..))")
    public void beforeAnyMethodInServicePackage() {
        System.out.println("Вызван метод в пакете service!");
    }
}
```

Здесь мы расширили наш Pointcut так, чтобы он реагировал на любой метод в любом классе в пакете service.

Пример 3: Применение аспекта к конкретным методам

```
@Aspect
@Component
public class MyAspect {

    @Before("execution(* com.example.service.MyService.specificMethod(..))")
    public void beforeSpecificMethod() {
        System.out.println("Вызван конкретный метод specificMethod!");
    }
}
```

В этом примере мы уточнили наш Pointcut, чтобы реагировать только на вызов метода specificMethod в классе MyService.

Использование Join Points через Pointcut выражения дает вам мощные инструменты для контроля над тем, где и когда ваши аспекты будут применяться. Это позволяет уточнять поведение вашего приложения, добавляя дополнительную функциональность там, где это необходимо.

Pointcut Expressions

Если вспомнить наш театральный пример, то Pointcut Expressions — это как режиссёрская инструкция: “Актёр А входит на сцену после фразы ‘Все было потеряно’”. Это уточнение показывает, когда именно должен произойти какой-то конкретный момент.

В мире Spring AOP Pointcut Expressions позволяют вам точно определить, на какие методы, классы или даже пакеты будет реагировать ваш аспект. Они являются частью магии AOP и предоставляют мощные инструменты для уточнения поведения вашего кода.

Структура Pointcut Expressions

Базовый формат Pointcut выглядит так:

execution(модификатор_доступа возвращаемый_тип имя_пакета.имя_класса.имя_метода(параметры))

Но большинство частей в этом формате являются опциональными. Давайте разберемся!

Пример 1: Выбор всех методов

```
@Pointcut("execution(* *.*(..))")
private void selectAllMethods() {}
```

Здесь * *.*(..) означает “любой метод любого класса с любыми аргументами”.

Пример 2: Выбор методов по имени

```
@Pointcut("execution(* *.set*(..))")
private void selectAllSetters() {}
```

Здесь мы используем шаблон set* для выбора всех методов, которые начинаются на “set”.

Пример 3: Выбор методов по параметрам

```
@Pointcut("execution(* *.find*(String))")
private void selectAllStringFinders() {}
```

Этот Pointcut выберет все методы, начинающиеся на “find” и принимающие один параметр типа String.

Дополнительные Pointcut выражения

Pointcut не ограничивается только модификатором execution. Есть и другие, например:

- within(): ограничивает матчинг методов в определенных классах или пакетах.

```
@Pointcut("within(com.example.service.*)")
private void allMethodsInService() {}
```

- this(): ограничивает матчинг методов, где прокси-объект имеет заданный тип.
- target(): ограничивает матчинг методов, где целевой объект (реальный объект, а не прокси) имеет заданный тип.
- args(): ограничивает матчинг методов, где аргументы имеют заданные типы.

```
@Pointcut("args(String,..)")
private void methodsWithStringFirstArg() {}
```

- @annotation(): реагирует на методы, которые имеют определенную аннотацию.

```
@Pointcut("@annotation(com.example.annotations.MyCustomAnnotation)")
private void methodsWithMyCustomAnnotation() {}
```

С помощью Pointcut Expressions у вас появляется гибкий и мощный инструмент, позволяющий детально контролировать, где именно будет применяться ваш аспект. Это позволяет вам создавать более чистый и модульный код, избегая повторений и улучшая читаемость.

Introductions

Что такое Introductions?

Давай начнём с понимания, что такое Introductions в контексте Spring AOP. Introductions, иногда называемые “Mixin”, позволяют вам добавить новые методы или поля к существующим классам. Они дают возможность объекту реализовывать интерфейс, который изначально он не реализовывал. Звучит сложно? На самом деле всё гораздо проще, чем кажется.

Аналогия

Представьте, что у вас есть игрушка — машинка. Эта машинка красиво ездит, но вам захотелось, чтобы она ещё и летала. Вместо того чтобы покупать новую игрушку, вы просто добавляете крылья к вашей машинке. Теперь ваша машинка не только ездит, но и летает.

Introductions в Spring AOP работают так же: они добавляют “крылья” вашим классам, расширяя их функциональность.

Пример кода с использованием Introductions

Допустим, у нас есть следующий интерфейс:

```
public interface Flyer {
    void fly();
}
```

И у нас есть класс Car, который, естественно, изначально не умеет летать:

```
public class Car {
    public void drive() {
        System.out.println("Car is driving...");
    }
}
```

Теперь, используя Introductions, мы можем “научить” Car летать:

```
@Aspect
public class FlyerIntroduction {

        @DeclareParents(value = "com.example.Car", defaultImpl =
FlyingImpl.class)
    public static Flyer flyer;
}

public class FlyingImpl implements Flyer {
    @Override
    public void fly() {
        System.out.println("Car is now flying!");
    }
}
```

Теперь каждый объект класса Car также будет реализовывать интерфейс Flyer, и вы сможете вызывать метод fly() для него!

Как это используется на практике?

Introductions могут быть полезны в ряде сценариев:

1. **Переиспользование кода:** Вместо дублирования методов в разных классах, вы можете использовать Introductions, чтобы предоставить общую функциональность.

2. **Постепенное внедрение новых возможностей:** Если вы хотите добавить новую функцию в ряд объектов, но не хотите изменять их базовый код, Introductions — ваш выбор.

3. **Работа с сторонними библиотеками:** Иногда у вас есть библиотека, и вы хотели бы добавить ей новые возможности, но у вас нет доступа к исходному коду. С помощью Introductions вы можете “расширить” функциональность этой библиотеки.

В целом, Introductions предоставляют дополнительный уровень гибкости при работе с объектами и могут быть невероятно полезными в реальной разработке, когда вы сталкиваетесь с необходимостью динамически расширять функциональность объектов.

AOP и обработка исключений

Друзья, в жизни каждого разработчика наступает момент, когда всё идет не так, как было запланировано. Иногда это может быть из-за исключений в коде. Но благодаря Spring AOP у нас есть инструменты, чтобы эффективно управлять и обрабатывать исключения, и в этом разделе мы рассмотрим, как это делать.

AfterThrowing: ваш первый шаг в обработке исключений

Advices AfterThrowing в Spring AOP позволяет нам “ловить” исключения, которые были выброшены во время выполнения метода. Этот Advices работает как блок catch в обычной обработке исключений.

Пример:

```
@Aspect
public class ErrorHandlingAspect {

    @AfterThrowing(pointcut = "execution(* com.example.*.*(..))", throwing = "error")
    public void handleAllErrors(Exception error) {
        System.out.println("An error occurred: " + error.getMessage());
        // Здесь можно, например, отправлять уведомление или логировать ошибку
    }
}
```

В примере выше, мы “ловим” все исключения, которые выбрасываются из методов пакета com.example, и выводим сообщение об ошибке.

Управление исключениями на уровне аспектов

Теперь, когда мы знаем, как “ловить” исключения с помощью AfterThrowing, давайте рассмотрим, как можно управлять этими исключениями.

1. **Логирование:** Чаще всего, при возникновении исключения, его логируют для дальнейшего анализа. AOP предоставляет удобный способ централизованного логирования всех исключений без необходимости вставлять блоки try-catch везде.
2. **Уведомления:** Можно настроить отправку уведомлений (например, на электронную почту или через мессенджер) при возникновении критических ошибок.
3. **Трансформация исключений:** Иногда нам нужно преобразовать одно исключение в другое. С помощью аспектов можно “перехватывать” конкретные исключения и выбрасывать другие, более информативные или специфичные для вашего приложения.
4. **Откат транзакций:** Если вы используете Spring для управления транзакциями, AOP может помочь вам автоматически откатывать транзакции при возникновении исключений.

Пример трансформации исключений:

```
@Aspect
public class ExceptionTransformingAspect {

        @AfterThrowing(pointcut = "execution(* com.example.service.*.*(..))",
throwing = "ex")
    public void transformExceptions(Exception ex) throws CustomException {
        if (ex instanceof SpecificException) {
            throw new CustomException("Custom message");
        }
    }
}
```

В этом примере, если возникает SpecificException, мы выбрасываем CustomException с кастомным сообщением. Это может быть полезно, если вы хотите

“скрыть” внутренние исключения вашего приложения и предоставить клиентам более общие или понятные ошибки.

Обработка исключений — критически важная часть любого приложения, и благодаря Spring AOP у нас есть мощные инструменты для эффективной работы с исключениями на высоком уровне, не усложняя основной код приложения.

Вспомним наши уроки по программированию. Если у вас есть несколько функций или методов, порядок их выполнения может иметь значение, верно? В мире AOP, когда у нас множество различных аспектов и Advices, которые могут влиять на один и тот же метод, порядок их выполнения становится критически важным. Итак, давайте разберёмся, как Spring управляет порядком выполнения Advices и как мы можем влиять на этот процесс.

Почему порядок так важен?

Допустим, у нас есть аспект, который логирует время выполнения метода, и другой аспект, который проверяет безопасность (например, права пользователя). В зависимости от порядка их выполнения результаты могут сильно отличаться. Если сначала срабатывает аспект безопасности и блокирует выполнение метода, то логирование времени выполнения может показывать нам неправильные данные. Так что порядок действительно имеет значение!

Как управлять порядком Advices в Spring AOP?

Spring предоставляет механизм, который называется “заказной номер” или Order. Каждый аспект может иметь свой заказной номер, который определяет его приоритет при выполнении. Чем меньше число, тем выше приоритет, и тем раньше будет выполнен аспект.

Пример:

Допустим, у нас есть два аспекта:

```
@Aspect
@Order(1)
public class SecurityAspect {
    // код аспекта безопасности
}

@Aspect
@Order(2)
public class LoggingAspect {
```

```
    // код аспекта логирования
}
```

В этом примере SecurityAspect будет выполнен перед LoggingAspect, потому что его Order равен 1, что меньше, чем у LoggingAspect.

Несколько Advices в одном аспекте

Если у нас есть несколько Advices в одном аспекте, они будут выполняться в порядке, определенном их типом. Так, например, Before Advices всегда выполняются перед After Advices. Но что, если у нас есть два Before Advicesa? Опять же, можно использовать Order для контроля их порядка выполнения.

Как видите, порядок выполнения Advices в Spring AOP — это мощный инструмент, который позволяет нам детально контролировать, как наши аспекты взаимодействуют с приложением. Используя аннотацию Order, мы можем легко управлять порядком выполнения наших аспектов и гарантировать, что наше приложение работает так, как мы этого ожидаем.

Прокси и AOP Proxy

Хорошо, друзья, мы уже немного освоились в мире AOP, но сейчас перед нами стоит еще одна дверь в глубины магии Spring’a. Внимание, сейчас будет очень интересно!

Как Spring создаёт прокси для AOP?

Когда мы говорим о применении аспектов к нашим компонентам в Spring, мы на самом деле говорим о создании “прокси”. Прокси — это своего рода “посредник” между вызывающим и целевым объектом. Spring создает эти прокси-объекты, чтобы вмешаться в вызов метода и выполнить нужные аспекты перед или после нашего основного метода.

Проще говоря, если у вас есть сервис, и вы хотите применить к нему логирование с помощью AOP, Spring создаст “фальшивую” копию вашего сервиса (прокси), которая будет выполнять логирование, а затем перенаправлять вызов на ваш реальный сервис.

JDK Dynamic Proxy vs. CGLIB proxy

Окей, так как же Spring создает эти прокси? Здесь у нас есть два основных механизма:

1. **JDK Dynamic Proxy:** Этот метод создает прокси для интерфейсов. Если ваш компонент или сервис реализует какой-либо интерфейс, Spring, как правило, использует этот метод. Под капотом здесь используется рефлексия Java, чтобы динамически создать новый объект, который реализует тот же интерфейс, что и ваш целевой объект.
2. **CGLIB proxy:** А что, если у вас класс, который не реализует никакого интерфейса? Здесь на помощь приходит CGLIB. Этот механизм создает подкласс вашего целевого класса. Внешне он выглядит и ведет себя как ваш оригинальный класс, но внутри он добавляет логику для аспектов.

Пример:

Допустим, у вас есть такой сервис:

```
public class SimpleService {
    public void doSomething() {
        // ваш код
    }
}
```

Если вы хотите применить к нему аспект, и используете CGLIB, Spring создаст что-то вроде:

```
public class SimpleService$$EnhancerBySpringCGLIB extends SimpleService {
    @Override
    public void doSomething() {
        // логика аспекта (например, логирование)
        super.doSomething();  // вызов оригинального метода
    }
}
```

А когда какой метод выбирать?

Spring автоматически решает, какой из механизмов использовать. Если ваш класс реализует интерфейс, Spring, скорее всего, будет использовать JDK Dynamic Proxy. Если нет интерфейса, то выбор падает на CGLIB. Но вы также можете явно указать Spring'у, какой механизм использовать, если у вас есть для этого особые причины.

Проксирование — это сердце магии AOP в Spring. Именно благодаря этим прокси-объектам Spring может применять наши аспекты к целевым компонентам.

Будь то JDK Dynamic Proxy или CGLIB, оба метода служат одной цели: делать ваш код чище и позволять разделить ответственность, используя аспекты.

Транзакции и AOP

Ага, здесь мы подходим к одной из самых важных и мощных особенностей Spring – управлению транзакциями с помощью AOP! Если вы работали с базами данных, вы, вероятно, слышали о транзакциях. Это механизмы, которые позволяют группировать несколько операций в одну целостную единицу работы. Если все операции проходят успешно, изменения сохраняются. Если хотя бы одна из операций завершается ошибкой, все изменения отменяются.

Так вот, в больших приложениях управление транзакциями может стать настоящим кошмаром. Но благодаря Spring и AOP этот процесс становится намного проще!

Основы управления транзакциями в Spring с использованием AOP

Spring предоставляет декларативное управление транзакциями, что означает, что вы можете управлять ими без написания специфического кода для этого. Вместо этого вы просто “объявляете” методы, которые должны выполняться в транзакции. И кто же помогает нам в этом? Наш старый друг AOP!

AOP обертывает вызов метода, делая его транзакционным. Если что-то идет не так внутри метода, AOP гарантирует, что транзакция откатывается.

@Transactional аннотация и её настройка

Самый простой способ объявить метод транзакционным в Spring - это использовать аннотацию @Transactional. Эта аннотация говорит Spring, что метод должен выполняться в контексте транзакции.

Пример:

```
@Service
public class BookService {

    @Autowired
    private BookRepository bookRepository;

    @Transactional
    public void addTwoBooks(Book book1, Book book2) {
```

```
        bookRepository.save(book1);
        bookRepository.save(book2);
    }
}
```

В приведенном примере, если сохранение book1 проходит успешно, но book2 по какой-то причине вызывает ошибку, то изменения, связанные с book1, будут автоматически отменены, и в базе данных ничего не изменится!

Вы также можете настроить поведение @Transactional, используя её параметры. Например, вы можете определить, при каких исключениях транзакция должна откатываться или даже установить разные уровни изоляции для транзакции.

Пример:

```
@Transactional(rollbackFor = CustomException.class)
public void someTransactionalMethod() {
    // ваш код
}
```

В этом примере транзакция будет откатываться только если будет выброшено исключение CustomException.

Управление транзакциями в Spring с AOP делает вашу жизнь разработчика намного проще. Вам не нужно беспокоиться о деталях управления транзакциями, вы просто объявляете свои намерения с помощью @Transactional, и Spring делает всю тяжелую работу за вас. Это мощный инструмент в вашем арсенале, так что используйте его с умом!

Перехватчики (Interceptors)

Если у вас возникает ощущение, что всё это начинает слегка напоминать шпионский триллер с множеством “перехватов” и “посыльных”, вы абсолютно правы! Разработка – это иногда настоящий детектив, где нам нужно изучать и следить за различными процессами. И сегодня мы узнаем больше о еще одном актере на этой сцене – перехватчиках, или Interceptors.

Отличие перехватчиков от Advices

Под “перехватчиками” в Spring мы понимаем механизмы, которые позволяют “перехватывать” входящие и исходящие сообщения или запросы, обычно в

контексте веб-приложений. Они часто используются для таких задач, как логирование, безопасность, производительность и другие перекрестные задачи.

Advices (Advices), с другой стороны, больше связаны с AOP и обеспечивают перекрестную функциональность на уровне метода.

В простых словах: - **Advices** — это действия, которые выполняются до, после или вокруг метода. - **Перехватчики** — это механизмы, которые “перехватывают” запросы, перед тем как они достигнут их назначения (например, контроллера в веб-приложении).

Примеры использования

Давайте рассмотрим пример перехватчика в Spring Web MVC:

```
public class LoggingInterceptor extends HandlerInterceptorAdapter {

    @Override
            public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("Request URL: " + request.getRequestURL().toString());
        return true;
    }

    @Override
             public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, ModelAndView
modelAndView) throws Exception {
        System.out.println("After handling the request");
    }

    @Override
            public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        System.out.println("Request completed");
    }
}
```

В этом примере у нас есть перехватчик, который регистрирует URL каждого входящего запроса, а также регистрирует информацию после обработки запроса и после завершения обработки запроса.

Чтобы этот перехватчик работал, его нужно зарегистрировать:

```
@Configuration
public class AppConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoggingInterceptor());
    }
}
```

С перехватчиками вы можете реализовать множество полезных функций, таких как: - Логирование всех входящих запросов. - Проверка аутентификации и авторизации. - Изменение ответов перед их отправкой пользователю. - И многое другое!

Заключение

Таким образом, перехватчики – это мощный инструмент для управления входящими и исходящими запросами в вашем веб-приложении, в то время как Advices AOP позволяют управлять поведением методов в вашем приложении. Оба эти механизма являются чрезвычайно полезными для реализации перекрестной функциональности.

Как AOP меняет правила игры в разработке

Друзья, давайте вспомним наш путь по миру AOP и попробуем осознать, как именно этот инструмент меняет всю картину разработки.

1. Повторное использование кода и модульность

При использовании Spring AOP мы можем выделить общие аспекты и переиспользовать их по всему приложению. Это особенно заметно, когда речь идет о логировании, безопасности, транзакциях и других перекрестных задачах. Вместо того чтобы вставлять один и тот же код в разные части приложения, мы создаем один аспект и говорим Spring, где и когда его применять.

2. Улучшение читаемости и поддержки кода

Без AOP ваш код может быть перегружен логикой, не относящейся напрямую к основной функциональности метода. С AOP мы выносим эту “побочную” логику в отдельные аспекты, делая основной код чище и проще для понимания.

3. Гибкая настройка

Помните наши Pointcut Expressions? Они предоставляют потрясающую гибкость в определении, когда и где применять аспекты. Это значит, что мы можем быстро и легко изменять поведение нашего приложения без необходимости переписывать основной код.

4. Обработка исключений

С помощью AOP мы можем легко управлять исключениями на уровне аспектов. Это дает нам возможность обрабатывать ошибки централизованно, упрощая отладку и предоставляя пользователям более информативные сообщения об ошибках.

5. Безупречные транзакции

Тема транзакций и AOP показала, как просто управлять транзакциями, определяя их границы с помощью аннотаций. Это делает код проще и устойчивым к ошибкам.

6. Понимание проксирования

С помощью нашего погружения в прокси и AOP Proxy мы узнали, как Spring создает прокси для реализации аспектов. Это знание может быть крайне полезным при оптимизации и отладке приложений.

AOP, безусловно, является одной из самых мощных особенностей Spring. Он предоставляет нам инструменты для реализации перекрестной функциональности на более высоком уровне, сохраняя при этом чистоту и модульность нашего кода. И, хотя на первый взгляд он может показаться сложным, надеюсь, что сегодняшний урок помог вам понять его лучше.

Спасибо за то, что присоединились к этому уроку. Надеюсь, ваши будущие проекты станут немного проще, благодаря AOP! Удачного кодинга!

От AOP к миру микросервисов

Друзья, мы успешно пройдены огромный путь по вселенной Spring AOP! Надеюсь, вы оценили, как мощная и гибкая система аспектов может улучшить ваш код, делая его чище, модульнее и, конечно же, проще в обслуживании.

С помощью AOP мы научились привносить изменения в наше приложение без вмешательства в основной код, что позволяет нам сохранять чистоту и сосредотачиваться на том, что действительно важно. А это очень важный урок, потому что в современном мире IT приложения становятся все более сложными и многофункциональными.

И это отличный момент для перехода к нашей следующей теме: **Spring Cloud и микросервисная архитектура**.

Когда приложения растут, их становится сложнее поддерживать в монолитной структуре. Именно тут на помощь приходят микросервисы! Вместо одного огромного приложения у нас будет множество небольших сервисов, каждый из которых отвечает за свою часть функциональности. Это похоже на то, как мы разделяем обязанности между разными аспектами в AOP, но на много большем масштабе.

Spring Cloud предоставляет нам инструменты, чтобы эти микросервисы могли взаимодействовать друг с другом, обеспечивая нам масштабируемость, отказоустойчивость и множество других возможностей, необходимых для создания современных, высоконагруженных приложений.

Так что готовьтесь к новому приключению! Микросервисы откроют перед нами новые горизонты и возможности в мире разработки. Надеюсь, что ваш опыт с AOP пригодится и поможет вам быстрее освоить все прелести микросервисной архитектуры.

До новых встреч на следующем уроке и благодарю за внимание!

Что можно почитать еще?

1. Изучение Spring Boot 2.0. Грег Тернквист

2. Освоение Spring Boot 2.0. Динеш Раджпут

Используемая литература

1. Spring Boot в действии. Крейг Уоллс.
2. Spring Microservices в действии. Джон Карнелл
3. Cloud Native Java. Джош Лонг и Кенни Бастани