Building Web Applications Using the Spring Framework

Aesthetic: Tarkeshwar Barua

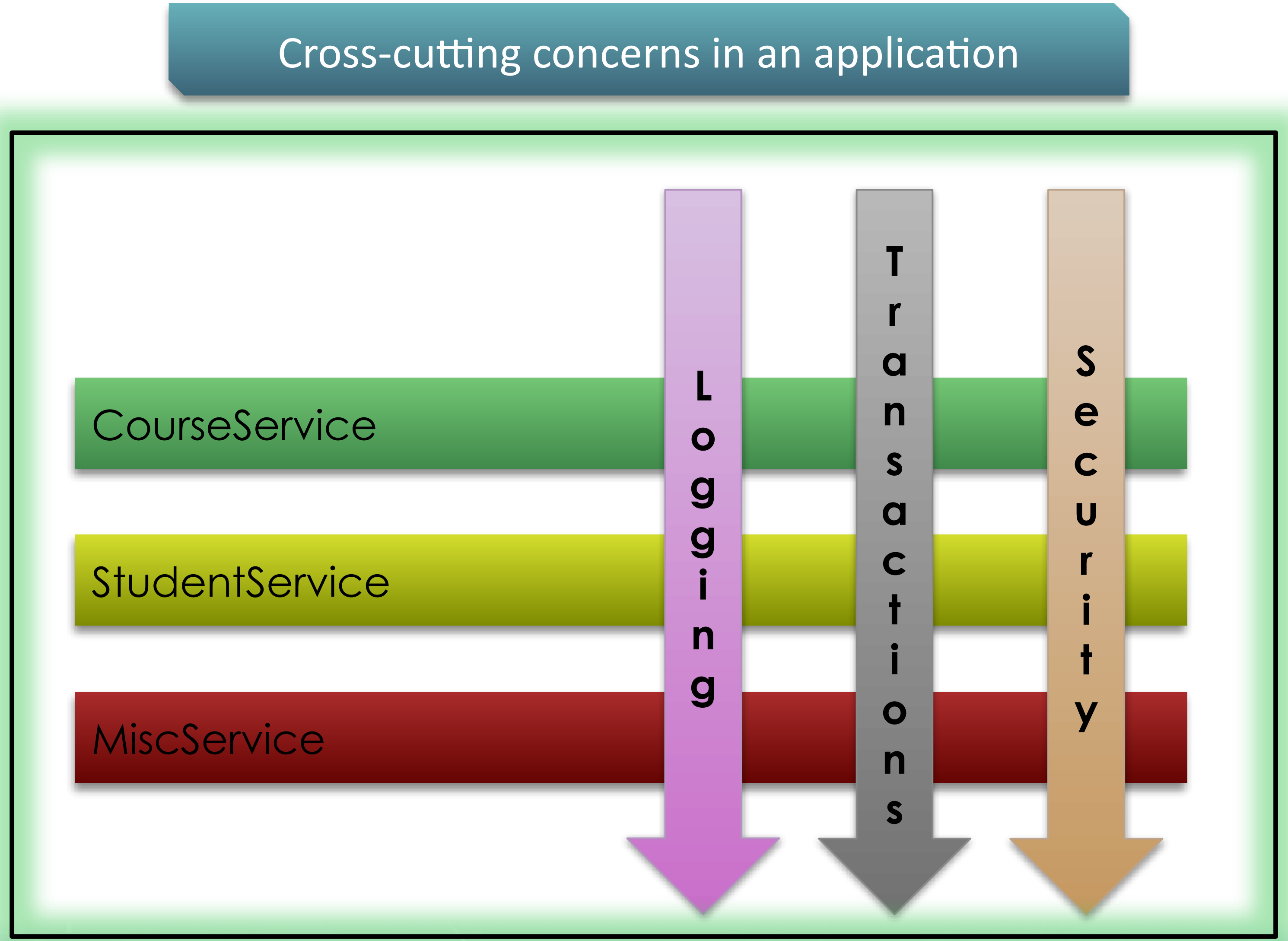
Introduction

- The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.
- Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns in AOP literature.)
- The Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

Where to use AOP

- Provides declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative transaction management.
- Allow users to implement custom aspects, complementing their use of OOP with AOP.
- The generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP.

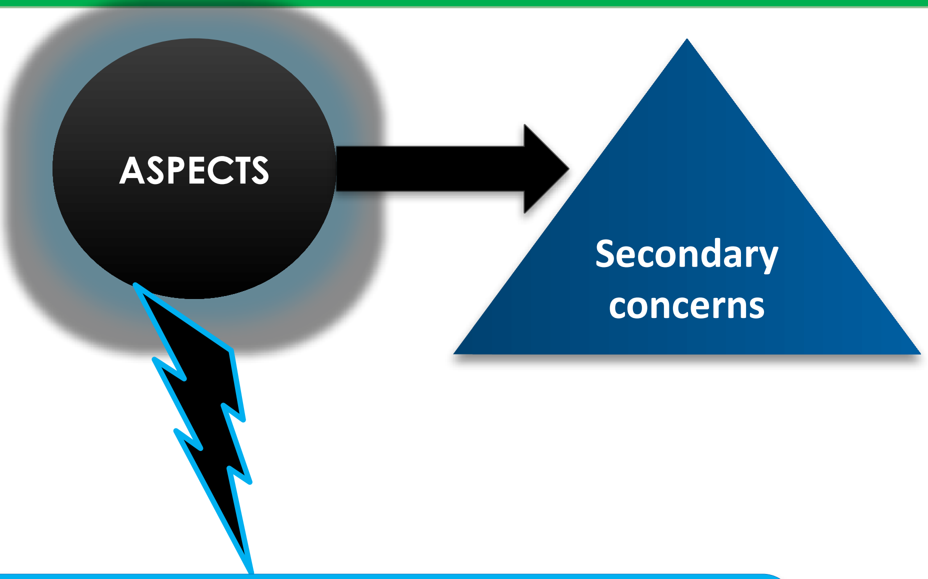
Introducing AOP



Introducing AOP (Contd.)

Spring provides Aspect-Oriented Programming (AOP).

AOP attempts to solve the problem of cross-cutting concerns by allowing you to express them in stand-alone modules called aspects.



Aspects enable you to isolate secondary logic from the primary business logic of the application.

Features of AOP

AOP

It increases modularity by isolating secondary logic from the primary logic.

It gives you the advantage of encapsulating the cross-cutting concerns.

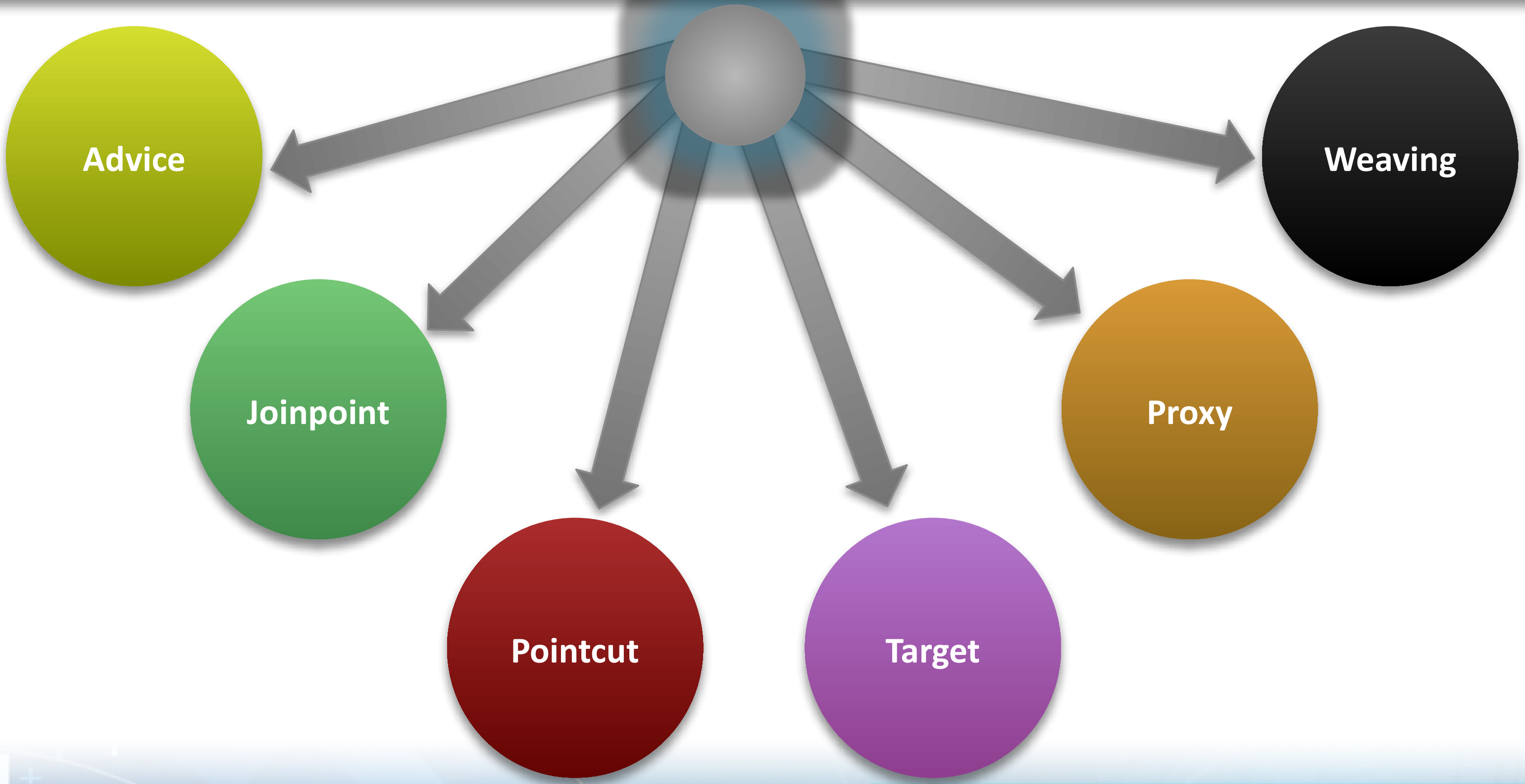
You can implement the aspects by defining methods in a Java class.

AOP allows easy removal of the previously defined functionalities without modifying the primary logic of the application.

Describing Aspects

The secondary concern of an application is considered as an aspect, such as login, security, authorization, and transaction management.

After identifying the aspect in an application, you need to implement AOP by identifying and creating the following components:



Aspect

- A modularization of a concern that cuts across multiple classes.
- Transaction management is a good example of a crosscutting concern in Java EE applications.
- In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the **@Aspect** annotation.
- It's an evolution of the decorator design pattern

Decorator Design Pattern

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
- We will decorate a shape with some color without alter shape class.

Creating Aspect

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {
@Before("execution(public String getName())")
	public void getNameAdvice(){
		System.out.println("Executing Advice on getName()");
	}

	@Before("execution(* com.tarkesh.spring.service.*.get*())")
	public void getAllAdvice(){
		System.out.println("Service method getter called");
	}
}
```

Join Point

- A point during the execution of a program, such as the execution of a method or the handling of an exception.
- In Spring AOP, a join point always represents a method execution.

Advice

- An action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice.
- Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.

Pointcut

- A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (The execution of a method with a certain name).
- The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

Creating Pointcut

- a signature comprising a name and any parameters, and a pointcut expression that determines exactly which method executions we are interested in
- **@AspectJ** annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the **@Pointcut** annotation

```
@Pointcut("execution(* transfer(..))")// the pointcut
expression
private void anyOldTransfer() {}// the pointcut signature
```

Point cut Designator

- **execution** - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
- **within** - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- **this** - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type

Point cut Designator

- **target** - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
- **args** - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
- **@target** - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type

Point cut Designator

- **@args** - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
- **@within** - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
- **@annotation** - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

pointcut expressions

- @Pointcut("execution(public * *(..))")
- private void anyPublicOperation() {}

  @Pointcut("within(com.xyz.someapp.trading..*)")
- private void inTrading() {}

- @Pointcut("anyPublicOperation() && inTrading()")
- private void tradingOperation() {}

Introduction

- declaring additional methods or fields on behalf of a type.
- Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object.
- You could use an introduction to make a bean implement an IsModified interface, to simplify caching.
- An introduction is known as an inter-type declaration in the AspectJ community.

Target Object

- object being advised by one or more aspects.
- Also referred to as the advised object.
- Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.

AOP Proxy

- an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on).
- In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

Weaving

- Linking aspects with other application types or objects to create an advised object.
- This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of Advice

- **Before advice:** Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- **After returning advice:** Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
- **After throwing advice:** Advice to be executed if a method exits by throwing an exception.

Types of Advice

- **After (finally) advice:** Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
- **Around advice:** Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Describing Aspects (Contd.)

Advice

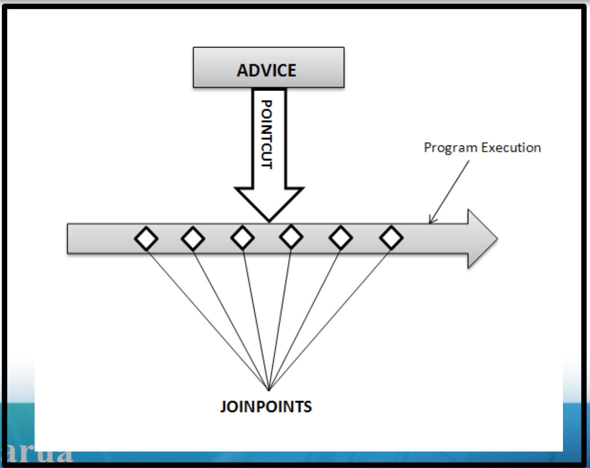
It is the action an aspect performs.

Joinpoint

It is a point or a location in the application, where an advice can be plugged in.

Pointcut

It defines to which joinpoint a particular advice should be applied.



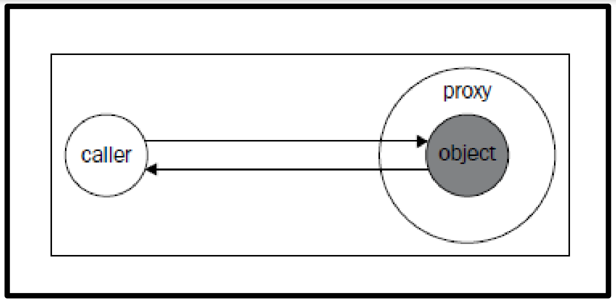
Describing Aspects (Contd.)

Target

Is an object to which the aspect is applied.

Proxy

Wraps the target object and intercepts all the calls made to the object in such a way that the calling object seems to be interacting with the target object rather than the proxy.



Weaving

Is the process of applying aspects to the target object at the specified joinpoint to create a new proxied object.

Just a Minute

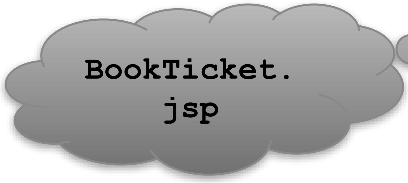
__________ defines to which joinpoint a particular advice should be applied.

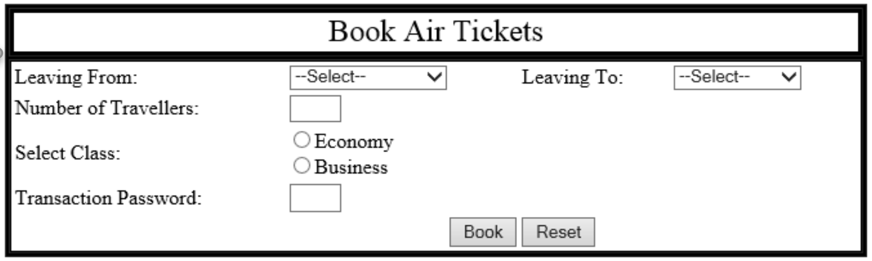
1. Pointcut
2. Joinpoint
3. Advice
4. Aspect

Answer: 1. Pointcut

Implementing AOP

Air ticket reservation application





Book Air Tickets

Leaving From:	--Select--	Leaving To:	--Select--
Number of Travellers:			
Select Class:	○ Economy ○ Business		
Transaction Password:			

Primary job

Book air tickets

The secondary jobs of the air ticket reservation application are considered as aspects.

To implement aspects, you need to perform the following operations:

Create advice

Define pointcut

Create proxy

Creating Advice

An advice is an action taken by the aspect at a particular joinpoint.

An application can have one or more advices.

Spring provides the following types of advices:

- Before
- After-returning
- After-throwing

Creating Advice (Contd.)

Before

[redacted]

```
public void before(Method method, Object[] args,Object target )
throws Throwable
```

After-returning

[redacted]

```
public void afterReturning(Object returnValue, Method method,
Object[] args,Object target) throws Throwable
```

After-throwing

[redacted]

```
public void afterThrowing(Throwable throwable)
```

Creating Advice (Contd.)

You can create advices in the following ways:

Using Java classes

Using configuration elements

Creating Advice (Contd.)

You can create a Java class to implement the required advices.

The following code snippet shows the implementation of before and after-returning advices in the `SecondaryJobAdvice` class:

Using Java classes

```
package AOP;
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;

public class SecondaryJobAdvice implements
MethodBeforeAdvice,AfterReturningAdvice {

    public SecondaryJob secondary;
    @Override
    public void before(Method method, Object[] args, Object target) throws
Throwable {
        secondary.authenticate(args[1].toString());
        secondary.checkSeatsAvailability(args[0].toString()); }
    @Override
    public void afterReturning(Object returnValue, Method method, Object[]
args, Object target) throws Throwable {
        secondary.updateSeats(args[0].toString());
        secondary.rewardGift(args[0].toString()); }
    public void setSecondary(SecondaryJob secondary) {
        this.secondary = secondary;}}
```

Creating Advice (Contd.)

The Spring configuration elements can be used to turn any Java class into an aspect by using the following `aop` namespace:

```
xmlns:aop="http://www.springframework.org/
schema/aop"
```

Using configuration elements

Spring provides the following configuration elements:

`<aop:config>`

`<aop:aspect>`

`<aop:before>`

`<aop:after-returning>`

`<aop:after-throwing>`

`<aop:around>`

Defining Pointcut

Spring provides the `org.springframework.aop.support.JdkRegexpMethodPointcut` class that allows you to define pointcuts by using regular expressions.

Pointcut is defined in the Spring configuration file.

```
<bean id="bookingPointCut"
class="org.springframework.aop.support.JdkRegexpMethod
Pointcut">
    <property name="pattern" value=".*book"/>
</bean>
```

A pattern, `.*book`, is specified in the `<property>` tag of the bean.

It means any method ending in book and belonging to any class in the application will be matched with the `value` attribute of the `pattern` property.

Defining Pointcut (Contd.)

Spring provides the `org.springframework.aop.support.DefaultPointcutAdvisor` class to create an advisor.

You need to define an advisor bean in the Spring configuration file.

```
<bean id="secondaryJobAdvisor"
class="org.springframework.aop.support.DefaultPointcut
Advisor">
    <property name="advice" ref="secondaryJobAdvice"/>
    <property name="pointcut" ref="bookingPointCut"/>
</bean>
```

Creating Proxy

To create a proxy bean, Spring provides the `org.springframework.aop.config.ProxyFactoryBean` class.

A proxy is created as a bean in the Spring configuration file.

```
<bean id="inst" class="AOP.BookTicket"/>
<bean id="bookProxy"
class="org.springframework.aop.framework.ProxyFactoryBe
an">
   <property name="target" ref="inst"/>
   <property name="interceptorNames"
value="secondaryJobAdvisor"/>
   <property name="proxyInterfaces"
value="AOP.BookTicketInterface"/>
</bean>
```

Parameters to the constructor of the `ProxyFactoryBean` class

Creating Proxy (Contd.)

The proxy needs to be called from the `HandleRequestController` class.

```
@RequestMapping(method=RequestMethod.POST)
    public String processView(@ModelAttribute("book")
BookTicketApp bt, ModelMap model) {
        ApplicationContext ctx=new
ClassPathXmlApplicationContext("AOP/Config.xml");
        BookTicketInterface
perf=(BookTicketInterface)ctx.getBean("bookProxy");
        perf.book(bt.getTravelers(), bt.getPassword());
…………………………
…………………………
    }
}
```

Just a Minute

While creating an advice, which one of the following configuration elements is used to define a method that is executed before and after a joinpoint?

1. `<aop:config>`
2. `<aop:around>`
3. `<aop:aspect>`
4. `<aop:before>`

Answer: 2. `<aop:around>`