UNIT II: Spring Boot: Creating a Spring Boot Application, Spring Boot Application Annotation, What is Autowiring , Scope of a bean, Logger, Introduction to Spring AOP, Implementing AOP advices, Best Practices : Spring Boot Application

What you will learn

- Understand Spring Boot and its advantages
- Develop Spring application using Spring Boot
- Develop application using the concept of Autowiring and bean scope
- Develop application using Aspect Oriented Programming

Spring 5 Basics with Spring Boot

Spring Framework is one of the most popularly used open-source frameworks for developing enterprise applications. Spring Core with Spring Boot course is designed to introduce the fundamental concepts of Spring Framework with Spring Boot. This course provides an overview of features, basic modules, and core concepts of Spring Framework using Spring Boot.

Pre Contents
- Java Programming Fundamentals
- Java Language Features

What is Spring Boot?

Spring Boot is a framework built on top of the Spring framework that helps the developers to build Spring-based applications very quickly and easily. The main goal of Spring Boot is to create Spring-based applications quickly without demanding developers to write the boilerplate configuration.

But how does it work? It works because of the following reasons,

1. Spring Boot is an opinionated framework

Spring Boot forms opinions. It means that Spring Boot has some sensible defaults which you can use to quickly build your application. For example, Spring Boot uses embedded Tomcat as the default web container.

2. Spring Boot is customizable

Though Spring Boot has its defaults, you can easily customize it at any time during your development based on your needs. For example, if you prefer log4j for logging over Spring Boot built-in logging support then you can easily make a

dependency change in your pom.xml file to replace the default logger with log4j dependencies.

The main Spring Boot features are as follows:

- Starter Dependencies
- Automatic Configuration
- Spring Boot Actuator
- Easy-to-use Embedded Servlet Container Support

Creating a Spring Boot Application

There are multiple approaches to create a Spring Boot application. You can use any of the following approaches to create the application:

- **Using Spring Boot CLI**
- **Using Spring Initializr**
- **Using the Spring Tool Suite (STS)**

 In this course, you will learn how to use Spring Initializr for creating Spring Boot applications.

It is an online tool provided by Spring for generating Spring Boot applications which is accessible at http://start.spring.io/. You can use it for creating a Spring Boot project using the following steps:

Step 1: Create your Spring Boot application launch Spring Initializr. You will get the following screen:

Note: This screen keeps changing depending on updates from Pivotal and changes in the Spring Boot version.

Step 2: Select Project as Maven, Language as Java, and Spring Boot as 2.1.13 and enter the project details as follows:

- Choose com.infy as Group
- Choose Demo_SpringBoot as Artifact

Click on More options and choose com.infy as Package Name

Step 3: Click on Generate Project. This would download a zip file to your local machine.

Step 4: Unzip the zip file and extract it to a folder.

Step 5: In Eclipse, Click File → Import → Existing Maven Project. Navigate or type in the path of the folder where you extracted the zip file to the next screen. After finishing, our Spring Boot project should look like as follows:
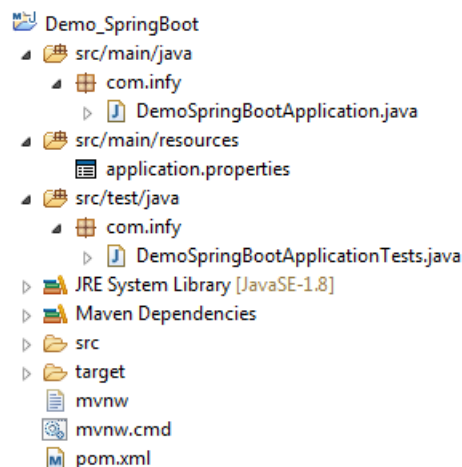
You have created a Spring Boot Maven-based project. Now let us explore what is contained in the generated project.

Understanding Spring Boot project structure

The generated project contains the following files:

1. pom.xml

This file contains information about the project and configuration details used by Maven to build the project.

```xml
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.  <modelVersion>4.0.0</modelVersion>
5.  <parent>
6.        <groupId>org.springframework.boot</groupId>
7.        <artifactId>spring-boot-starter-
   parent</artifactId>
8.        <version>2.1.13.RELEASE</version>
9.        <relativePath/> <!-- lookup parent from
   repository -->
10.       </parent>
11.       <groupId>com.infy</groupId>
12.       <artifactId>Demo_SpringBoot</artifactId>
13.       <version>0.0.1-SNAPSHOT</version>
14.       <name>Demo_SpringBoot</name>
15.       <description>Demo project for Spring
   Boot</description>
16.       <properties>
17.           <java.version>1.8</java.version>
18.       </properties>
19.       <dependencies>
20.           <dependency>
21.
   <groupId>org.springframework.boot</groupId>
22.                   <artifactId>spring-boot-
   starter</artifactId>
23.           </dependency>
24.           <dependency>
25.
   <groupId>org.springframework.boot</groupId>
26.                   <artifactId>spring-boot-starter-
   test</artifactId>
27.                   <scope>test</scope>
28.           </dependency>
29.       </dependencies>
30.       <build>
31.           <plugins>
32.                   <plugin>
33.
   <groupId>org.springframework.boot</groupId>
```

```
34.                          <artifactId>spring-boot-maven-
   plugin</artifactId>
35.                     </plugin>
36.              </plugins>
37.        </build>
38. </project>

39.
```

2. application.properties

This file contains application-wide properties. To configure your application Spring reads the properties defined in this file. In this file, you can define a server's default port, the server's context path, database URLs, etc.

3. DemoSpringBootApplication.java

```
1. @SpringBootApplication
2. public class DemoSpringBootApplication {
3.     public static void main(String[] args) {
4.         SpringApplication.run(DemoSpringBootApplication
   .class, args);
5.     }
6. }

7.
```

It is annotated with @SpringBootApplication annotation which triggers auto-configuration and component scanning and can be used to declare one or more @Bean methods also. It contains the main method which bootstraps the application by calling the run() method on the SpringApplication class. The run method accepts DemoSpringBootApplication.class as a parameter to tell Spring Boot that this is the primary component.

4. DemoSpringBootApplicationTest.java

In this file test cases are written. This class is by default generated by Spring Boot to bootstrap Spring application.

Spring Boot Starters

Spring Boot starters are pre-configured dependency descriptors with the most commonly used libraries that you can add to your application. So you don't need to search for compatible libraries and configure them manually. Spring Boot will ensure that the necessary libraries are added to the build. To use these starters, you have to add them to the pom.xml file. For example, to use spring-boot-starter following dependency needs to be added in pom.xml:

```
1. <dependency>
2.     <groupId>org.springframework.boot</groupId>
3.     <artifactId>spring-boot-starter</artifactId>
4. </dependency>

5.
```

Spring Boot comes with many starters. Some popular starters which we are going to use in this course are as follows:

- spring-boot-starter - This is the core starter that includes support for auto-configuration, logging, and YAML.
- spring-boot-starter-aop - This starter is used for aspect-oriented programming with Spring AOP and AspectJ.
- spring-boot-starter-data-jdbc - This starter is used for Spring Data JDBC.
- spring-boot-starter-data-jpa - This starter is used for Spring Data JPA with Hibernate.
- spring-boot-starter-web - This starter is used for building a web application using Spring MVC and Spring REST. It also provides Tomcat as the default embedded container.
- spring-boot-starter-test - This starter provides support for testing Spring Boot applications using libraries such as JUnit, Hamcrest, and Mockito.

Spring Boot Starter Parent

The Spring Boot Starter Parent defines key versions of dependencies and default plugins for quickly building Spring Boot applications. It is present in the pom.xml file of the application as a parent as follows:

```
1. <parent>
2.     <groupId>org.springframework.boot</groupId>
3.     <artifactId>spring-boot-starter-
   parent</artifactId>
4.     <version>2.1.13.RELEASE</version>
5.     <relativePath/>
```

```
6. </parent>

7.
```

It allows you to manage the following things for multiple child projects and modules:

- **Configuration** – The Java version and other properties.
- **Dependencies** – The version of dependencies.
- **Default Plugins Configuration** – This includes default configuration for Maven plugins such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, maven-war-plugin.

Executing the Spring Boot application

To execute the Spring Boot application run the DemoSpringBootApplication as a standalone Java class which contains the main method. On successful execution you will get the following output in the console:

Spring Boot Runners

So far you have learned how to create and start the Spring Boot application. Now suppose you want to perform some action immediately after the application has started then for this Spring Boot provides the following two interfaces:

- CommandLineRunner
- ApplicationRunner

CommandLineRunner is the Spring Boot interface with a run() method. Spring Boot automatically calls this method of all beans implementing this interface after the application context has been loaded. To use this interface, you can modify the DemoSpringBootApplication class as follows:

```
1.      @SpringBootApplication
```

```
2.      public class DemoSpringBootApplication
implements CommandLineRunner {
3.          public static void main(String[] args) {
4.
  SpringApplication.run(DemoSpringBootApplication.class,
args);
5.          }
6.          @Override
7.          public void run(String... args) throws
Exception {
8.              System.out.println("Welcome to
CommandLineRunner");
9.          }
10.    }
```

Spring Boot application is configured using a file named application.properties. It is auto detected without any Spring based configurations and is placed inside the "src/main/resources" directory as shown below:

```
Demo_SpringBoot
  src/main/java
    com.infy
      DemoSpringBootApplication.java
  src/main/resources
    application.properties
  src/test/java
    com.infy
      DemoSpringBootApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```
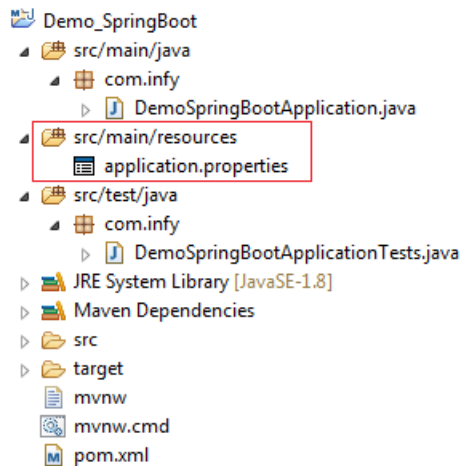
In this file, various default properties are specified to support logging, AOP, JPA, etc. All the default properties need not be specified in all cases. We can specify them only on-demand. At startup, the Spring application loads all the properties and adds them to the Spring Environment class.

To use a custom property add it to the application.properties file.

application.properties

```
1. message= Welcome Spring
```

Then autowire the Environment class into a class where the property is required.

```
1. @Autowired

2.  Environment env;
```

You can read the property from Environment using the getProperty() method.

```
1. env.getProperty("message")
```

You can use other files to keep the properties. For example, InfyTelmessage.properties.

InfyTelmessage.properties

```
1. message=Welcome To InfyTel
```

But by default Spring Boot will load only the application.properties file. So how you will load the **InfyTelmessage.properties** file?

In Spring @PropertySource annotation is used to read from properties file using Spring's Environment interface. The location of the properties file is mentioned in the Spring configuration file using @PropertySource annotation.

So InfyTelmessage.properties which are present in classpath can be loaded using @PropertySource as follows:

```
1.
2. import
   org.springframework.context.annotation.PropertySource;
3.
4. @SpringBootApplication
5. @PropertySource("classpath:InfyTelmessage.properties")
```

```
 6. public class DemoSpringBootApplication {
 7.
 8.   public static void main(String[] args) throws
    Exception {
 9.         //code
10.         }
11. }

12.
```

To read the properties you need to autowire the Environment class into a class where the property is required

```
1. @Autowired

2. Environment env;
```

You can read the property from Environment using the getProperty() method.

```
1. env.getProperty("message")
```

Understanding @SpringBootApplication Annotation

We have already learned that the class which is used to bootstrap the Spring Boot application is annotated with @SpringBootApplication annotation as follows:

```
1. @SpringBootApplication
2. public class DemoSpringBootApplication {
3.     public static void main(String[] args) {
4.         SpringApplication.run(DemoSpringBootApplication
   .class, args);
5.     }
6. }

7.
```

Now let us understand this annotation in detail.

The **@SpringBootApplication** annotation indicates that it is a configuration class and also triggers auto-configuration and component scanning. It is a combination of the following annotations with their default attributes:

@EnableAutoConfiguration – This annotation enables auto-configuration for the Spring boot application which automatically configures our application based on the dependencies that you have added.

@ComponentScan – This enables the Spring bean dependency injection feature by using @Autowired annotation. All application components which are annotated   with @Component, @Service, @Repository,   or @Controller are automatically registered as Spring Beans. These beans can be injected by using @Autowired annotation.

@Configuration – This enables Java based configurations for Spring boot application.

The class that is annotated with @SpringBootApplication will be considered as the main class, is also a bootstrap class. It kicks starts the application by invoking the SpringApplication.run() method. You need to pass the .class file name of your main class to the run() method.

@SpringBootApplication is a convenience annotation that adds all of the following:

- @Configuration: Tags the class as a source of bean definitions for the application context.
- @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
  For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
- @ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.
- The main() method uses Spring Boot's SpringApplication.run() method to launch an application.

SpringBootApplication- scanBasePackages

By default, SpringApplication scans the configuration class package and all it's sub-packages. So if our SpringBootApplication class is in "com.eta" package, then it won't scan com.infy.service or com.infy.repository package. We can fix this situation using the SpringBootApplication scanBasePackages property.

```
1.      package com.eta;
```

```
2.        @SpringBootApplication(scanBasePackages={"com.in
fy.service","com.infy.repository"})
3.
4.     public class DemoSpringBootApplication {
5.         public static void main(String[] args) {
6.             SpringApplication.run(DemoSpringBootAppl
ication.class, args);
7.          }
8.      }
```

Demo on Spring Boot

Highlights:

Objective: To understand the Spring IOC feature using SpringBoot

Demo Steps:

Demo5Application .java

```
1. package com.infy;
2.
3. import org.springframework.boot.SpringApplication;
4. import
   org.springframework.boot.autoconfigure.SpringBootApplicatio
   n;
5. import
   org.springframework.context.support.AbstractApplicationCont
   ext;
6.
7. import com.infy.service.CustomerServiceImpl;
8.
9. @SpringBootApplication
10. public class Demo5Application{
11.
12.     public static void main(String[] args) {
13.
```

```
14.                    CustomerServiceImpl service = null;
15.                    AbstractApplicationContext context =
   (AbstractApplicationContext) SpringApplication
16.
   .run(Demo5Application.class, args);
17.                    service = (CustomerServiceImpl)
   context.getBean("customerService");
18.
   System.out.println(service.fetchCustomer());
19.              context.close();
20.         }
21.
22. }

23.
```

CustomerService.java

```
1. package com.infy.service;
2.
3. public interface CustomerService {
4.   public String fetchCustomer();
5.   public String createCustomer();
6. }

7.
```

CustomerServiceImpl.java

```
1. package com.infy.service;
2.
3. import org.springframework.beans.factory.annotation.Value;
4. import org.springframework.stereotype.Service;
5.
6.
7. @Service("customerService")
8. public class CustomerServiceImpl implements CustomerService
   {
9.
10.       @Value("10")
11.       private int count;
12.
13.       public String fetchCustomer() {
```

```
14.              return " The no of customers fetched are : "
   + count;
15.         }
16.
17.      public String createCustomer() {
18.              return "Customer is successfully created";
19.         }
20.
21. }

22.
```

OUTPUT:

```
1.
2.        .    ____
3.       /\\ / ___'_ __ _ _(_)_ __ __ _  \ \ \ \
4.      ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
5.       \\/  ___)| |_)| | | | | | || (_| |  ) ) ) )
6.        '  |____| .__|_| |_|_| |_\__, | / / / /
7.       =========|_|==============|___/=/_/_/_/
8.       :: Spring Boot ::        (v2.1.13.RELEASE)
9.
10.      2020-04-07 17:21:31.908  INFO 119940 --- [       main]
   com.infy.Demo5Application               : Starting Demo5Application on
11.      2020-04-07 17:21:31.912  INFO 119940 --- [       main]
   com.infy.Demo5Application               : No active profile set, falling
   back to default profiles:
12.      2020-04-07 17:21:34.211  INFO 119940 --- [       main]
   com.infy.Demo5Application               : Started Demo5Application in
   3.998 seconds
13.      The no of customers fetched are : 10
```

Summary: Spring Boot

This module helped us in learning the following:

- What is the Spring Boot?
- How to create a Spring Boot application?

What is Autowiring ?

n Spring if one bean class is dependent on another bean class then the bean dependencies need to be explicitly defined in your configuration class. But you can let the Spring IoC container to inject the dependencies into dependent bean

classes without been defined in your configuration class. This is called as autowiring.

To do autowiring, you can use @Autowired annotation. This annotation allows the Spring IoC container to resolve and inject dependencies into your bean. @Autowired annotation performs **byType Autowiring** i.e. dependency is injected based on bean type. It can be applied to attributes, constructors, setter methods of a bean class.

Autowiring is done only for dependencies to other beans. It doesn't work for properties such as primitive data types, String, Enum, etc. For such properties, you can use the **@Value** annotation.

Autowiring the property

```
@Autowired
private CustomerRepository customerRepository;
```

- @Autowired is specified on the property of the bean to be autowired.
- Java reflection will be used for initializing this property.

Autowiring the setter method

```
@Autowired
setCustomerRepository(CustomerRepository customerRepository){
this.customerRepository = customerRepository;
}
```

- @Autowired is specified at the setter method of the property to be autowired.
- The setter method will be called for Autowiring the property.

Autowiring the constructor

```
@Autowired
CustomerService(CustomerRepository customerRepository){
this.customerRepository = customerRepository;
}
```

- @Autowired is specified at the parameterized constructor.
- Parameterized constructor will be called for Autowiring the property byType.

Now let us see how to use @Autowired annotation.

@Autowired on Setter methods

The @Autowired annotation can be used on setter methods. This is called a Setter Injection.

```
1. package com.infy.service;
2. public class CustomerServiceImpl implements
   CustomerService {
3.
4.   private CustomerRepository customerRepository;
5.
6.   @Autowired
7.   public void setCustomerRepository(CustomerRepository
   customerRepository) {
```

```
8.          this.customerRepository = customerRepository;
9.   }
10.       --------

11. }
```

In the above code snippet, the Spring IoC container will call the setter method for injecting the dependency of CustomerRepository.

@Autowired on Constructor

The @Autowired annotation can also be used on the constructor. This is called a Constructor Injection.

```
1. package com.infy.service;
2. public class CustomerServiceImpl implements CustomerService {
3.
4.    private CustomerRepository customerRepository;
5.
6.    @Autowired
7.    public CustomerServiceImpl(CustomerRepository customerRepository) {
8.          this.customerRepository = customerRepository;
9.    }
10.      --------------

11.}
```

@Autowired on Properties

Let us now understand the usage of @Autowired on a property in Spring.

We will use @Autowired in the below code to wire the dependency of CustomerService class for CustomerRepository bean dependency.

```
1. package com.infy.service;
2. public class CustomerServiceImpl {
3. // CustomerService needs to contact CustomerRepository,
   hence injecting the customerRepository dependency
4.    @Autowired
5.    private CustomerRepository customerRepository;
6. ------------

7. }
```

@Autowired is by default wire the dependency based on the type of bean.

In the above code, the Spring container will perform dependency injection using the Java Reflection API. It will search for the class which implements CustomerRepository and injects its object. The dependencies which are injected using @Autowired should be available to the Spring container when the dependent bean object is created. If the container does not find a bean for autowiring, it will throw the NoSuchBeanDefinitionException exception.

If more than one beans of the same type are available in the container, then the framework throws an exception indicating that more than one bean is available for autowiring. To handle this **@Qualifier** annotation is used as follows:

```
1. package com.infy.service;
2. public class CustomerServiceImpl {
3.
4.  @Autowired
5.       @Qualifier("custRepo")
6.  private CustomerRepository customerRepository;
7.          ------------
8. }
```

In Spring @Value annotation is used to insert values into variables and method arguments. Using @Value we can either read spring environment variables or system variables.

We can assign a default value to a class property with @Value annotation:

```
1. public class CustomerDTO {
2.  @Value("1234567891")
3.  long phoneNo;
4.  @Value("Jack")
5.  String name;
6.  @Value("Jack@xyz.com")
7.  String email;
8.  @Value("ANZ")
9.  String address;

10. }
```

Note that it accepts only a String argument but the passed-in value gets converted to an appropriate type during value-injection.

To read a Spring environment variable we can use @Value annotation:

```
1. public class CustomerDTO {
2.   @Value("${value.phone}")
3.   long phoneNo;
4.   @Value("${value.name}")
5.   String name;
6.   @Value("${value.email}")
7.   String email;
8.   @Value("${value.address}")
9.   String address;
10. }
```

Demo On @Autowiring Spring Boot

Highlights:

Objective: To understand the Autowiring in Spring

Demo Steps:

Demo6Application .java

```
1. package com.infy;
2.
3. import java.util.List;
4. import org.springframework.boot.SpringApplication;
5. import
   org.springframework.boot.autoconfigure.SpringBootApplic
   ation;
6. import
   org.springframework.context.support.AbstractApplication
   Context;
7. import com.infy.dto.CustomerDTO;
8. import com.infy.service.CustomerServiceImpl;
9.
10. @SpringBootApplication
```

```
11. public class Demo6Application {
12.
13.     public static void main(String[] args) {
14.
15.         CustomerServiceImpl service = null;
16.         AbstractApplicationContext context =
   (AbstractApplicationContext)
   SpringApplication.run(Demo6Application.class,
17.                         args);
18.         service = (CustomerServiceImpl)
   context.getBean("customerService");
19.         List<CustomerDTO> listcust =
   service.fetchCustomer();
20.         System.out.println("PhoneNumer" + "    " +
   "Name" + "     " + "Email" + "      " + "Address");
21.         for (CustomerDTO customerDTO2 : listcust)
   {
22.             System.out.format("%5d%10s%20s%10s",
   customerDTO2.getPhoneNo(), customerDTO2.getName(),
23.                         customerDTO2.getEmail(),
   customerDTO2.getAddress());
24.             System.out.println();
25.         }
26.     }
27.
28. }

29.
```

CustomerService.java

```
1. package com.infy.service;
2.
3. import java.util.List;
4.
5. import com.infy.dto.CustomerDTO;
6.
7. public interface CustomerService {
8.   public String createCustomer(CustomerDTO customerDTO);
9.   public List<CustomerDTO> fetchCustomer();
10.
11. }
```

```
12.
```

CustomerServiceImpl.java

```java
1. package com.infy.service;
2.
3. import java.util.List;
4. import
   org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.stereotype.Service;
6. import com.infy.dto.CustomerDTO;
7. import com.infy.repository.CustomerRepository;
8.
9. @Service("customerService")
10. public class CustomerServiceImpl implements
    CustomerService{
11.
12.         @Autowired
13.         private CustomerRepository customerRepository;
14.
15.         public String createCustomer(CustomerDTO
    customerDTO) {
16.
      customerRepository.createCustomer(customerDTO);
17.
18.             return "Customer with " +
    customerDTO.getPhoneNo() + " added successfully";
19.         }
20.
21.         public List<CustomerDTO> fetchCustomer() {
22.             return customerRepository.fetchCustomer();
23.         }
24. }

25.
```

CustomerRepository.java

```java
1. package com.infy.repository;
2.
```

```java
3.
4. import java.util.ArrayList;
5. import java.util.List;
6.
7. import javax.annotation.PostConstruct;
8.
9. import org.springframework.stereotype.Repository;
10. import com.infy.dto.CustomerDTO;
11.
12.
13. @Repository
14. public class CustomerRepository {
15.     List<CustomerDTO> customers = null;
16.
17. //Equivalent/similar to constructor. Here, populates
    the DTOs in a hard-coded way
18.     @PostConstruct
19.     public void initializer()
20.     {
21.         CustomerDTO customerDTO = new
    CustomerDTO();
22.         customerDTO.setAddress("Chennai");
23.         customerDTO.setName("Jack");
24.         customerDTO.setEmail("Jack@infy.com");
25.         customerDTO.setPhoneNo(9951212221);
26.         customers = new ArrayList<>();
27.         customers.add(customerDTO);
28.     }
29.
30.     //adds the received customer object to customers
    list
31.     public void createCustomer(CustomerDTO
    customerDTO)
32.     {
33.         customers.add(customerDTO);
34.     }
35.
36.     //returns a list of customers
37.     public List<CustomerDTO> fetchCustomer()
38.     {
39.         return customers;
40.     }
41. }
```

```
42.
```

CustomerDTO.java

```java
1. package com.infy.dto;
2.
3. public class CustomerDTO {
4.   long phoneNo;
5.   String name;
6.   String email;
7.   String address;
8.
9.   public long getPhoneNo() {
10.           return phoneNo;
11.       }
12.
13.       public void setPhoneNo(long phoneNo) {
14.           this.phoneNo = phoneNo;
15.       }
16.
17.       public String getName() {
18.           return name;
19.       }
20.
21.       public void setName(String name) {
22.           this.name = name;
23.       }
24.
25.       public String getEmail() {
26.           return email;
27.       }
28.
29.       public void setEmail(String email) {
30.           this.email = email;
31.       }
32.
33.       public String getAddress() {
34.           return address;
35.       }
36.
37.       public void setAddress(String address) {
38.           this.address = address;
```

```
39.          }
40.
41.      public CustomerDTO(long phoneNo, String name,
   String email, String address) {
42.              this.phoneNo = phoneNo;
43.              this.name = name;
44.              this.email = email;
45.              this.address = address;
46.
47.      }
48.      public CustomerDTO() {
49.
50.      }
51. }

52.
```

OUTPUT:

```
1.
2.    .   ____          _            __ _ _
3.   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
4.  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
5.   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
6.    '  |____| .__|_| |_|_| |_\__, | / / / /
7.  =========|_|==============|___/=/_/_/_/
8.  :: Spring Boot ::        (v2.1.13.RELEASE)
9.
10. 2020-04-07 12:11:56.259  INFO 78268 --- [
   main] com.infy.Demo6Application            :
11. 2020-04-07 12:11:56.263  INFO 78268 --- [
   main] com.infy.Demo6Application            :
12. 2020-04-07 12:11:57.137  INFO 78268 --- [
   main] com.infy.Demo6Application            :
13. PhoneNumer    Name        Email         Address
14. 9951212222   Jack    Jack@infy.com   Chennai
```

Scope of a bean

The lifetime of a bean depends on its scope. Bean's scope can be defined while declaring it in the configuration metadata file.

A bean can be in singleton or prototype scope. A bean with the "**singleton**" scope is initialized during the container starts up and the same bean instance is provided for every bean request from the application. However, in the case of the **"prototype"** scope, a new bean instance is created for every bean request from the application.

Let us now understand about bean scope in detail.

There will be a single instance of "singleton" scope bean in the container and the same bean is given for every request from the application.

The bean scope can be defined for a bean using @Scope annotation in Java class. By default, the scope of a bean is the singleton

```
1.      package com.infy.service;
2.
3.      @Service("customerService")
4.      @Scope("singleton")
5.      public class CustomerServiceImpl implements CustomerService {
6.              @Value("10")
7.              private int count;
8.
9.              public int getCount() {
10.                     return count;
11.             }
12.
13.             public void setCount(int count) {
14.                     this.count = count;
15.             }
16.
17.             public String fetchCustomer() {
18.                     return " The number of customers fetched are : "
+ count;
19.             }
20.
21.     }
```

For the "prototype" bean, there will be a new bean created for every request from the application.

In the below example, customerService bean is defined with prototype scope. There will be a new customerService bean created for every bean request from the application.

```
1.      package com.infy.service;
2.
3.      import
org.springframework.beans.factory.annotation.Value;
4.      import
org.springframework.context.annotation.Scope;
```

```
5.        import org.springframework.stereotype.Service;
6.
7.        @Service("customerService")
8.        @Scope("prototype")
9.        public class CustomerServiceImpl implements
CustomerService {
10.               @Value("10")
11.               private int count;
12.
13.               public int getCount() {
14.                       return count;
15.               }
16.
17.               public void setCount(int count) {
18.                       this.count = count;
19.               }
20.
21.               public String fetchCustomer() {
22.                       return " The number of customers
fetched are : " + count;
23.               }
24.
25.       }
```

Demo: Scope of a bean Spring Boot

Highlights:

Objective: To understand the Prototype scope of a bean

Demosteps:

CustomerService.java

```
1. package com.infy.service;
2.
3. public interface CustomerService {
4.
5.   public String fetchCustomer();
6.
```

```
7. }
```

CustomerServiceImpl.java

```java
1. package com.infy.service;
2.
3. @Service("customerService")
4. @Scope("prototype")
5. public class CustomerServiceImpl implements CustomerService
   {
6.   @Value("10")
7.   private int count;
8.
9.   public int getCount() {
10.             return count;
11.         }
12.
13.         public void setCount(int count) {
14.             this.count = count;
15.         }
16.
17.         public String fetchCustomer() {
18.             return " The number of customers fetched are
   : " + count;
19.         }
20.
21. }

22.
```

Demo7Application.java

```java
1. package com.infy;
2.
3. import org.springframework.boot.SpringApplication;
4. import
   org.springframework.boot.autoconfigure.SpringBootApplicatio
   n;
5. import
   org.springframework.context.support.AbstractApplicationCont
   ext;
6. import com.infy.service.CustomerServiceImpl;
7.
```

```
 8. @SpringBootApplication
 9. public class Demo7Application {
10.
11.       public static void main(String[] args) {
12.             AbstractApplicationContext context =
   (AbstractApplicationContext)
   SpringApplication.run(Demo7Application.class,
13.                    args);
14.             CustomerServiceImpl service1 =
   (CustomerServiceImpl) context.getBean("customerService");
15.             System.out.println("The customerservice1
   output=" + service1.fetchCustomer());
16.             service1.setCount(20);
17.             System.out.println("The customerservice1
   output after setmethod=" + service1.fetchCustomer());
18.             CustomerServiceImpl service2 =
   (CustomerServiceImpl) context.getBean("customerService");
19.             System.out.println("The customerservice2
   output =" + service2.fetchCustomer());
20.             System.out.println(service1==service2);
21.             context.close();
22.       }
23.
24. }

25.
```

Output

```
1.       /\\ / ___'_ __ _ _(_)_ __ __ _ \ \ \ \
2.      ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
3.       \\/ ___)| |_)| | | | | || (_| |  ) ) ) )
4.        '  |____| .__|_| |_|_| |_\__, | / / / /
5.       ========|_|==============|___/=/_/_/_/
6.       :: Spring Boot ::        (v2.1.13.RELEASE)
7.
8.      2020-04-07 17:20:51.414  INFO 18992 --- [
   main] com.infy.Demo7Application              :
9.      2020-04-07 17:20:51.417  INFO 18992 --- [
   main] com.infy.Demo7Application              : 2020-
   04-07 17:20:52.438  INFO 18992 --- [        main]
   com.infy.Demo7Application              :
10.     The customerservice1 output= The number of
   customers fetched are : 10
```

```
11.     The customerservice1 output after setmethod= The
number of customers fetched are : 20
12.     The customerservice2 output = The number of
customers fetched are : 10
13.     false
```

Summary: Autowiring and Scope of a bean

This module helped us in learning the following:

- What is Autowiring?
- Different types of Autowiring
- What are the scope of a bean and its different types?

SpringBoot Logging

Logger

What is logging?

Logging is the process of writing log messages to a central location during the execution of the program. That means Logging is the process of tracking the execution of a program, where

- Any event can be logged based on the interest to the
- When exception and error occurs we can record those relevant messages and those logs can be analyzed by the programmer later

There are multiple reasons why we may need to capture the application activity.

- Recording unusual circumstances or errors that may be happening in the program
- Getting the info about what's going in the application

There are several logging APIs to make logging easier. Some of the popular ones are:

- JDK Logging API
- Apache Log4j
- Commons Logging API

The Logger is the object which performs the logging in applications.

Levels in the logger specify the severity of an event to be logged. The logging level is decided based on necessity. For example, TRACE can be used during development and ERROR during deployment.

The following table shows the different levels of logging.

Level	Description
ALL	For all the levels (including user defined levels)
TRACE	Informational events
DEBUG	Information that would be useful for debugging the application
INFO	Information that highlights the progress of an application
WARN	Potentially harmful situations
ERROR	Errors that would permit the application to continue running
FATAL	Severe errors that may abort the application
OFF	To disable all the levels

You know that logging is one of the important activities in any application. It helps in quick problem diagnosis, debugging, and maintenance. Let us learn the logging configuration in Spring Boot.

While executing your Spring Boot application, have you seen things like the below getting printed on your console?

```
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2017-07-26 11:33:42.344  INFO 5624 --- [           main] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManagerFactory for persistence unit 'default'
2017-07-26 11:33:42.442  INFO 5624 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [
        name: default
        ...]
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate Core {5.0.12.Final}
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.cfg.Environment            : HHH000206: hibernate.properties not found
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.cfg.Environment            : HHH000021: Bytecode provider name : javassist
2017-07-26 11:33:42.932  INFO 5624 --- [           main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
2017-07-26 11:33:44.703  INFO 5624 --- [           main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
2017-07-26 11:33:45.936  INFO 5624 --- [           main] org.hibernate.tool.hbm2ddl.SchemaUpdate  : HHH000228: Running hbm2ddl schema update
2017-07-26 11:33:46.247  INFO 5624 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
```

Do you have any guess what are these?

Yes, you are right. These are logging messaged logged on the **INFO level.** However, you haven't written any code for logging in to your application. Then who does this?

By default, Spring Boot configures logging via **Logback** to log the activities of libraries that your application uses.

As a developer, you may want to log the information that helps in quick problem diagnosis, debugging, and maintenance. So, let us see how to customize the default logging configuration of Spring Boot so that your application can log the information that you are interested in and in your own format.

Have you realized that you have not done any of the below activities for logging which you typically do in any Spring application?

- Adding dependent jars for logging
- Configuring logging through Java configuration or XML configuration

Still, you are able to log your messages. The reason is Spring Boot's default support for logging. The spring-boot-starter dependency includes spring-boot-starter-logging dependency, which configures logging via Logback to log to the console at the INFO level.

Spring Boot uses Commons Logging API with default configurations for Java Util Logging, Log4j 2, and Logback implementation. Among these implementations, Logback configuration will be enabled by default.

You, as a developer, have just created an object for Logger and raise a request to log with your own message in LoggingAspect.java as shown below.

```
1. public class CustomerServiceImpl implements
   CustomerService
2. {
3. private static Logger logger =
   LoggerFactory.getLogger(CustomerServiceImpl.class);
4.
5. public void deleteCustomer(long phoneNumber) {
6.
7.        public void deleteCustomer(long phoneNumber) {
8.        try {
9.
   customerRepository.deleteCustomer(phoneNumber);
10.              } catch (Exception e) {
11.                    logger.info("In log Exception ");
12.                    logger.error(e.getMessage(),e);
13.              }
14.
15.        }

16. }
```

Apart from info(), the Logger class provides other methods for logging information:

Method	Description
void **debug**(Object msg)	Logs messages with the Level DEBUG
void **error**(Object msg)	Logs messages with the Level ERROR
void **fatal**(Object msg)	Logs messages with the Level FATAL
void **info**(Object msg)	Logs messages with the Level INFO
void **warn**(Object msg)	Logs messages with the Level WARN
void **trace**(Object msg)	Logs messages with the Level TRACE
void **debug**(Object msg)	Logs messages with the Level DEBUG

The default log output contains the following information.

Date and Time

```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect   : CustomerService.USERID_ALREADY_EXIST
```

Log level

```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect   : CustomerService.USERID_ALREADY_EXIST
```

Process id

```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect   : CustomerService.USERID_ALREADY_EXIST
```

Thread name

```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect   : CustomerService.USERID_ALREADY_EXIST
```

Separator

```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect   : CustomerService.USERID_ALREADY_EXIST
```

Logger name

```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Log message

```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

But, how to change this default configuration if you want to,

- log the message in a file rather than console
- log the message in your own pattern
- log the messages of a specific level
- use Log4j instead of Logback

Log into file

By default Spring Boot logs the message on the console. To log into a file, you have to include either logging.file or logging. path property in your application.properties file.

Note: Please note that from Spring boot 2.3.X version onwards logging.file and logging.path has been deprecated we should use "logging.file.name" and " logging.file.path" for the same.

Custom log pattern

Include logging.pattern.* property in application.properties file to write the log message in your own format.

Logging property	Sample value	Description
logging.pattern.console	%d{yyyy-MM-dd HH:mm:ss,SSS}	Specifies the log pattern to use on the console
logging.pattern.file	%5p [%t] %c [%M] - %m%n	Specifies the log pattern to use in a file

Custom log level

By default, the logger is configured at the INFO level. You can change this by configuring the logging.level.* property in application.properties file as shown below.

```
1. logging.level.root=WARN
2. logging.level.com.infosys.ars=ERROR

3.
```

Log4j instead of Logback

Since Spring Boot chooses Logback implementation by default, you need to exclude it and then include log4j 2 instead of in your pom.xml.

```
1.      <dependency>
2.          <groupId>org.springframework.boot</groupId>
3.          <artifactId>spring-boot-starter</artifactId>
4.          <exclusions>
5.              <exclusion>
6. <groupId>org.springframework.boot</groupId>
7.                  <artifactId>spring-boot-starter-logging</artifactId>
8.              </exclusion>
9.          </exclusions>
10.     </dependency>
11.     <dependency>
12.         <groupId>org.springframework.boot</groupId>
13.         <artifactId>spring-boot-starter-log4j2</artifactId>
14.     </dependency>
```

Demo: Logger

Highlights:

Objective: To implement Logging

Demo Steps:

CustomerService.java

```
1. package com.infy.service;
2.
3. import com.infy.dto.CustomerDTO;
4.
5. public interface CustomerService {
6.
7.   public String createCustomer(CustomerDTO dto);
8.
9.   public String fetchCustomer();
10.
11.       public void deleteCustomer(long phoneNumber)
    throws Exception;

12. }
```

CustomerServiceImpl.java

```
1. package com.infy.service;
2.
3. import org.slf4j.Logger;
4. import org.slf4j.LoggerFactory;
5. import
   org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.stereotype.Service;
7.
8. import com.infy.dto.CustomerDTO;
9. import com.infy.repository.CustomerRepository;
10.
11. @Service("customerService")
12.
13. public class CustomerServiceImpl implements
    CustomerService {
14.       private static Logger logger =
    LoggerFactory.getLogger(CustomerServiceImpl.class);
15.       @Autowired
16.       private CustomerRepository customerRepository;
17.
18.
19.
20.       @Override
21.       public String createCustomer(CustomerDTO dto) {
```

```
22.            return
   customerRepository.createCustomer(dto);
23.
24.        }
25.
26.        @Override
27.        public String fetchCustomer() {
28.
29.            return customerRepository.fetchCustomer();
30.        }
31.
32.    @Override
33.      public void deleteCustomer(long phoneNumber) {
34.
35.            try {
36.
   customerRepository.deleteCustomer(phoneNumber);
37.            } catch (Exception e) {
38.                logger.info("In log Exception ");
39.                logger.error(e.getMessage(),e);
40.            }
41.
42.        }
43. }

44.
```

CustomerRepository.java

```
1. package com.infy.repository;
2.
3.
4. import java.util.ArrayList;
5. import java.util.List;
6.
7. import javax.annotation.PostConstruct;
8.
9. import org.springframework.stereotype.Repository;
10.
11. import com.infy.dto.CustomerDTO;
12.
13.
14. @Repository
```

```java
15. public class CustomerRepository {
16.         @PostConstruct
17.         public void initializer()
18.         {
19.                 CustomerDTO customerDTO = new
    CustomerDTO();
20.                 customerDTO.setAddress("Chennai");
21.                 customerDTO.setName("Jack");
22.                 customerDTO.setEmail("Jack@infy.com");
23.                 customerDTO.setPhoneNo(9951212221);
24.                 customers = new ArrayList<>();
25.                 customers.add(customerDTO);
26.         }
27.         List <CustomerDTO> customers=null;
28.         public String createCustomer(CustomerDTO dto) {
29.                 customers = new ArrayList<>();
30.                 customers.add(dto);
31.                 return "Customer added
    successfully"+customers.indexOf(dto);
32.         }
33.
34.         public String fetchCustomer()  {
35.                 return " The customer fetched
    "+customers;
36.         }
37.         public void deleteCustomer(long phoneNumber)
    throws Exception
38.         {
39.
40.                 for(CustomerDTO customer : customers)
41.                 {
42.                         if(customer.getPhoneNo() ==
    phoneNumber)
43.                         {
44.                                 customers.remove(customer);
45.
    System.out.println(customer.getName()+"of
    phoneNumber"+customer.getPhoneNo()+"\t got deleted
    successfully");
46.                                 break;
47.                         }
48.                         else
49.                                 throw new Exception("Customer
    does not exist");
```

```
50.                    }
51.
52.            }
53. }

54.
```

CustomerDTO.java

```java
1. package com.infy.dto;
2.
3. public class CustomerDTO {
4.   long phoneNo;
5.   String name;
6.   String email;
7.   String address;
8.
9.   public long getPhoneNo() {
10.            return phoneNo;
11.        }
12.
13.       public void setPhoneNo(long phoneNo) {
14.            this.phoneNo = phoneNo;
15.        }
16.
17.       public String getName() {
18.            return name;
19.        }
20.
21.       public void setName(String name) {
22.            this.name = name;
23.        }
24.
25.       public String getEmail() {
26.            return email;
27.        }
28.
29.       public void setEmail(String email) {
30.            this.email = email;
31.        }
32.
33.       public String getAddress() {
34.            return address;
```

```
35.         }
36.
37.        public void setAddress(String address) {
38.                this.address = address;
39.        }
40.
41.        public CustomerDTO(long phoneNo, String name,
   String email, String address) {
42.                this.phoneNo = phoneNo;
43.                this.name = name;
44.                this.email = email;
45.                this.address = address;
46.
47.        }
48.
49.        public CustomerDTO() {
50.
51.        }
52. }

53.
```

Demo8Application.java

```
1. package com.infy;
2.
3. import org.springframework.boot.SpringApplication;
4. import
   org.springframework.boot.autoconfigure.SpringBootApplic
   ation;
5. import
   org.springframework.context.support.AbstractApplication
   Context;
6. import com.infy.service.CustomerServiceImpl;
7.
8. @SpringBootApplication
9. public class Demo8Application {
10.
11.        public static void main(String[] args) {
12.
13.                CustomerServiceImpl service = null;
```

```
14.            AbstractApplicationContext context =
   (AbstractApplicationContext)
   SpringApplication.run(Demo8Application.class,
15.                     args);
16.
17.            service = (CustomerServiceImpl)
   context.getBean("customerService");
18.
19.            service.deleteCustomer(11512122221);
20.            // service.deleteCustomer(99512122221);
21.        }
22.
23. }

24.
```

Output:

```
1.
2.   .   ____          _            __ _ _
3.  /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
4. ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
5.  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
6.   '  |____| .__|_| |_|_| |_\__, | / / / /
7.  =========|_|==============|___/=/_/_/_/
8.  :: Spring Boot ::        (v2.1.13.RELEASE)
9.
10. 2020-04-07 14:50:12.615  INFO 99756 --- [
   main] com.infy.Demo8Application              :
   Starting Demo8Application
11. 2020-04-07 14:50:12.621  INFO 99756 --- [
   main] com.infy.Demo8Application              : No
   active profile set,
12. 2020-04-07 14:50:14.183  INFO 99756 --- [
   main] com.infy.Demo8Application              :
   Started Demo8Application in
13. 2020-04-07 14:50:14.187  INFO 99756 --- [
   main] com.infy.service.CustomerServiceImpl   : In log
   Exception
14. 2020-04-07 14:50:14.192 ERROR 99756 --- [
   main] com.infy.service.CustomerServiceImpl   :
   Customer does not exist
15.
```

```
16. java.lang.Exception: Customer does not exist
17.      at
    com.infy.repository.CustomerRepository.deleteCustomer(C
    ustomerRepository.java:49) ~[classes/:na]
18.      at
    com.infy.service.CustomerServiceImpl.deleteCustomer(Cus
    tomerServiceImpl.java:38) ~[classes/:na]
19.      at
    com.infy.Demo8Application.main(Demo8Application.java:19
    ) [classes/:na]
```

Summary: Spring Boot Logging

This module helped us in learning the following:

- What is Logging?
- How to implement logging in Spring Boot?

Introduction to Spring AOP (See Appendix A)

AOP (Aspect Oriented Programming) is used for applying common behaviors like transactions, security, logging, etc. to the application.

These common behaviors generally need to be called from multiple locations in an application. Hence, they are also called as cross cutting concerns in AOP.

Spring AOP provides the solution to cross cutting concerns in a modularized and loosely coupled way.
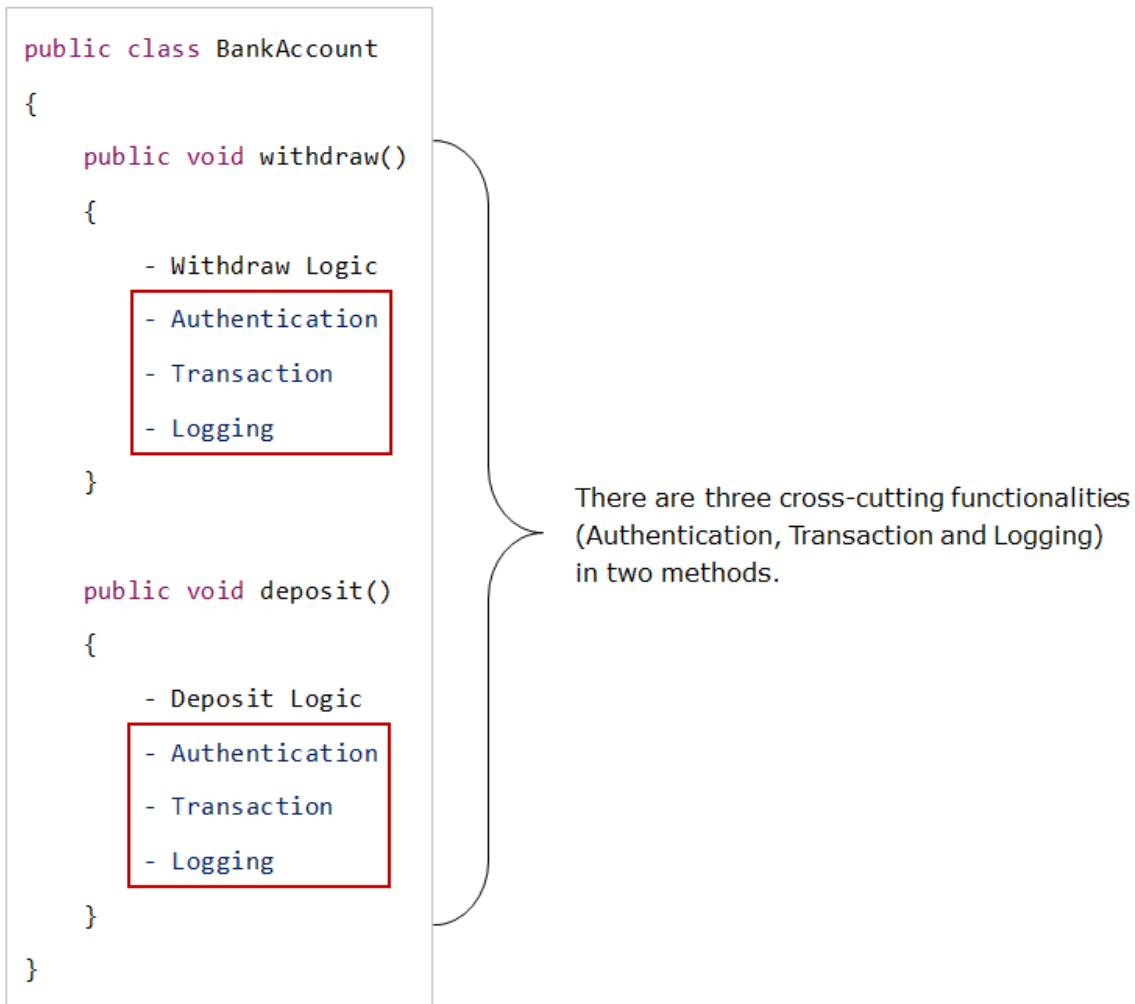
Advantages

- AOP ensures that cross cutting concerns are kept separate from the core business logic.
- Based on the configurations provided, the Spring applies cross cutting concerns appropriately during the program execution.
- This allows creating a more loosely coupled application wherein you can change the cross cutting concerns code without affecting the business code.
- In Object Oriented Programming(OOP), the key unit of modularity is class. But in AOP the key unit of modularity is an Aspect.

What is an Aspect?

Aspects are the cross-cutting concerns that cut across multiple classes.
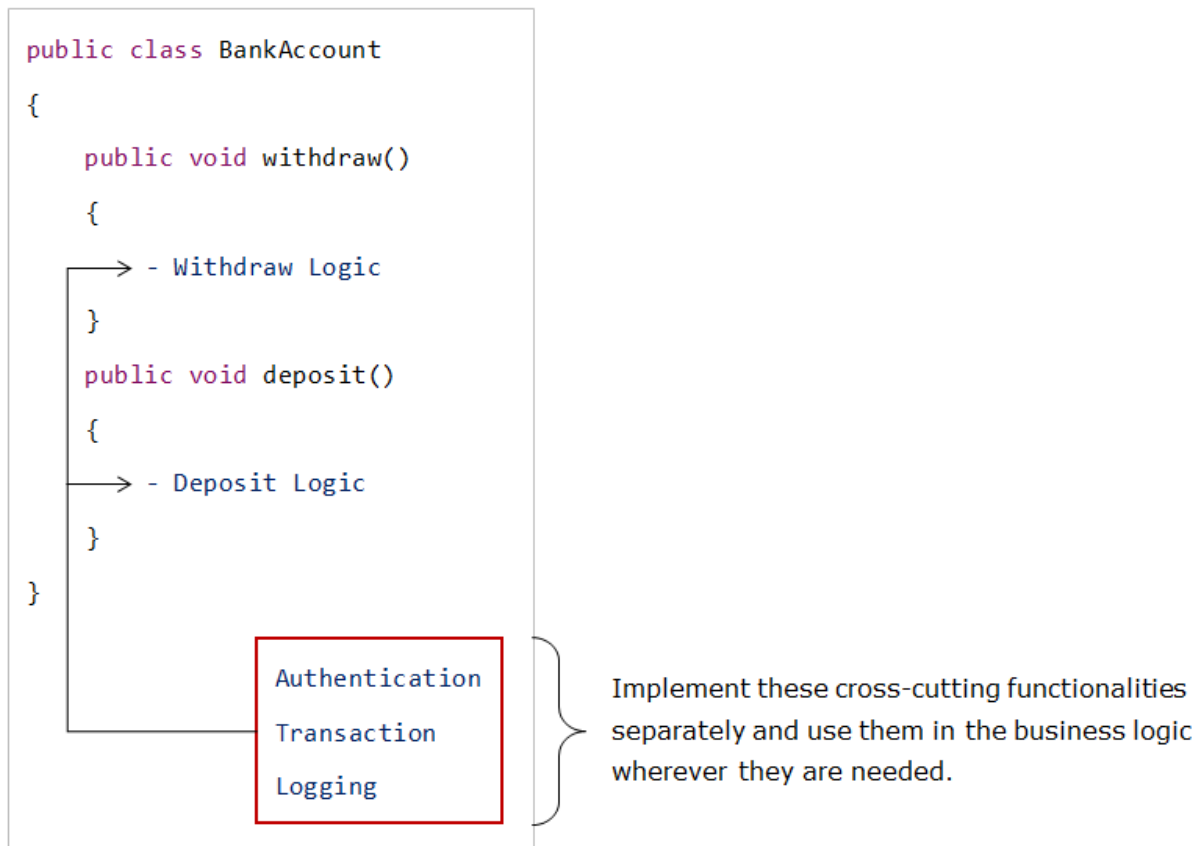
Examples: Transaction Management, Logging, Security, etc.

For a better understanding of Aspect Oriented Programming(AOP) concepts let us consider a Banking scenario comprising of BankAccount class with Withdraw and Deposit functionalities as shown below.

```
public class BankAccount
{
    public void withdraw()
    {
        - Withdraw Logic
        - Authentication
        - Transaction
        - Logging
    }


    public void deposit()
    {
        - Deposit Logic
        - Authentication
        - Transaction
        - Logging
    }
}
```

There are three cross-cutting functionalities (Authentication, Transaction and Logging) in two methods.

In a single class, cross-cutting functionalities are repeated twice. Think of a bigger scenario with many classes. You might need to repeat the cross-cutting concerns many times.

In Spring AOP, we can add these cross-cutting concerns at run time by separating the cross-cutting concerns from the client logic as shown below.

```
public class BankAccount
{
    public void withdraw()
    {
        ────────► - Withdraw Logic
    }
    public void deposit()
    {
        ────────► - Deposit Logic
    }
}
```

Authentication

Transaction

Logging

Implement these cross-cutting functionalities separately and use them in the business logic wherever they are needed.

In Spring AOP, we can add the cross-cutting functionalities at run time by separating the system services (cross-cutting functionalities) from the client logic.

- Aspect is a class that implements cross-cutting concerns. To declare a class as an Aspect it should be annotated with @Aspect annotation. It should be applied to the class which is annotated with @Component annotation or with derivatives of it.
- Joinpoint is the specific point in the application such as method execution, exception handling, changing object variable values, etc. In Spring AOP a join point is always the execution of a method.
- Advice is a method of the aspect class that provides the implementation for the cross-cutting concern. It gets executed at the selected join point(s). The following table shows the different types of advice along with the execution point they have

Type Of Execution	Execution Point
Before	Before advice is executed before the Joinpoint execution.
After	After advice will be executed after the execution of Joinpoint whether it returns with or without exception. Similar to finally block in exception

Type Of Execution	Execution Point
	handling.
AfterReturning	AfterReturning advice is executed after a Joinpoint executes and returns successfully without exceptions
AfterThrowing	AfterThrowing advice is executed only when a Joinpoint exits by throwing an exception
Around	Around advice is executed around the Joinpoints which means Around advice has some logic which gets executed before Joinpoint invocation and some logic which gets executed after the Joinpoint returns successfully

- Pointcut represents an expression that evaluates the method name before or after which the advice needs to be executed.
- In Spring AOP, we need to modularize and define each of the cross cutting concerns in a single class called Aspect.
- Each method of the Aspect which provides the implementation for the cross cutting concern is called Advice.
- The business methods of the program before or after which the advice can be called is known as a Joinpoint.
- The advice does not get inserted at every Joinpoint in the program.
- An Advice gets applied only to the Joinpoints that satisfy the Pointcut defined for the advice.
- Pointcut represents an expression that evaluates the business method name before or after which the advice needs to be called.

Consider an Aspect "LoggingAspect" as shown below to understand different Spring AOP terminologies. LoggingAspect is defined as Spring AOP Aspect by using @Aspect annotation.

A pointcut is an important part of AOP. So let us look at pointcut in detail.

Pointcut expressions have the following syntax:

```
1. execution(<modifiers> <return-type> <fully qualified
   class name>.<method-name>(parameters))

2.
```

where,

- execution is called a pointcut designator. It tells Spring that joinpoint is the execution of the matching method.
- <modifiers> determines the access specifier of the matching method. It is not mandatory and if not specified defaults to the public.
- <return-type> determines the return type of the method in order for a join point to be matched. It is mandatory. If the return type doesn't matter wildcard * is used.
- <fully qualified class name> specifies the fully qualified name of the class which has methods on the execution of which advice gets executed. It is optional. You can also use * wildcard as a name or part of a name.
- <method-name> specifies the name of the method on the execution of which advice gets executed. It is mandatory. You can also use * wildcard as a name or part of a name.

- parameters are used for matching parameters. To skip parameter filtering, two dots(..) are used in place of parameters.

Pointcut	Description
execution(public * *(..))	execution of any public methods
execution(* service *(..))	execution of any method with a name beginning with "service"
execution(*com.infy.service.CustomerServiceImpl.*(..))	execution of any method defined in CustomerServiceImpl of com.infy.service package
execution(* com.infy.service.*.*(..))	execution of any method defined in the com.infy.service package
execution(public * com.infy.service.CustomerServiceImpl.*(..))	execution of any public method of CustomerServiceImpl of com.infy.service package
execution(public String com.infy.service.CustomerserviceImpl.*(..))	execution of all public method of CustomerServiceImpl of com.infy.service package that returns a String

To use Spring AOP and AspectJ in the Spring Boot project you have to add the spring-boot-starter-aop starter in the pom.xml file as follows:

```
1. <dependency>
2.     <groupId>org.springframework.boot</groupId>
3.     <artifactId>spring-boot-starter-aop</artifactId>
4. </dependency>
```

Implementing AOP advices

Before Advice:

This advice is declared using @Before annotation. It is invoked before the actual method call. ie. This advice is executed before the execution of fetchCustomer()methods of classes present in com.infy.service package. The following is an example of this advice:

```
1. @Before("execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ")
2.  public void logBeforeAdvice(JoinPoint joinPoint) {
3.       logger.info("In Before Advice, Joinpoint
   signature :{}", joinPoint.getSignature());
4.       long time = System.currentTimeMillis();
5.       String date =
   DateFormat.getDateTimeInstance().format(time);
6.       logger.info("Report generated at time:{}",
   date);

7. }
```

After Advice:

This advice is declared using @After annotation. It is executed after the execution of the actual method(fetchCustomer), even if it throws an exception during execution. It is commonly used for resource cleanup such as temporary files or closing database connections. The following is an example of this advice :

```
1. @After("execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ")
2.  public void logAfterAdvice(JoinPoint joinPoint) {
3.       logger.info("In After Advice, Joinpoint
   signature :{}", joinPoint.getSignature());
4.       long time = System.currentTimeMillis();
5.       String date =
   DateFormat.getDateTimeInstance().format(time);
6.       logger.info("Report generated at time {}",
   date);
7.   }
```

After Returning Advice

This advice is declared using @AfterReturning annotation. It gets executed after joinpoint finishes its execution. If the target method throws an exception the advice is not executed. The following is an example of this advice that is executed after the method execution of fetchCustomer()method of classes present in com.infy.service package.

```
1. @AfterReturning(pointcut = "execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ")
2.  public void logDetails(JoinPoint joinPoint) {
3.       logger.info("In AfterReturning Advice, Joinpoint
   signature :{}", joinPoint.getSignature());

4. }
```

You can also access the value returned by the joinpoint by defining the returning attribute of @AfterReturning annotation as follows:

```
1. @AfterReturning(pointcut = "execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ", returning = "result")
2.  public void logDetails(JoinPoint joinPoint, String
   result) {
3.       logger.info("In AfterReturning Advice with
   return value, Joinpoint signature :{}",
   joinPoint.getSignature());
4.       logger.info(result.toString());

5. }
```

In the above code snippet, the value of the returning attribute is returnvalue which matches the name of the advice method argument.

AfterThrowing Advice :

This advice is defined using @AfterThrowing annotation. It gets executed after an exception is thrown from the target method. The following is an example of this advice that gets executed when exceptions are thrown from the fetchCustomer() method of classes present in com.infy.service package. So it is marked with @AfterThrowing annotation as follows:

```
1. @AfterThrowing("execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ")
2.  public void logAfterThrowingAdvice(JoinPoint
   joinPoint) {
```

```
3.        logger.info("In After throwing Advice, Joinpoint
   signature :{}", joinPoint.getSignature());
4.


5.}
```

You can also access the exception thrown from the target method inside the advice method as follows:

```
1. @AfterThrowing(pointcut ="execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ",throwing = "exception")
2.  public void logAfterThrowingAdvice(JoinPoint
   joinPoint,Exception exception) {
3.        logger.info("In After throwing Advice, Joinpoint
   signature :{}", joinPoint.getSignature());
4.        logger.info(exception.getMessage());

5.  }
```

Demo: AOP

Highlights:

Objective: To implement AOP

Demo Steps:

LoggingAspect.java

```
1. package com.infy.util;
2.
3. import java.text.DateFormat;
4. import java.util.List;
5.
6. import org.aspectj.lang.JoinPoint;
7. import org.aspectj.lang.ProceedingJoinPoint;
8. import org.aspectj.lang.annotation.After;
9. import org.aspectj.lang.annotation.AfterReturning;
```

```java
10. import org.aspectj.lang.annotation.AfterThrowing;
11. import org.aspectj.lang.annotation.Around;
12. import org.aspectj.lang.annotation.Aspect;
13. import org.aspectj.lang.annotation.Before;
14. import org.slf4j.Logger;
15. import org.slf4j.LoggerFactory;
16. import org.springframework.stereotype.Component;
17.
18. import com.infy.dto.CustomerDTO;
19.
20. @Component
21. @Aspect
22. public class LoggingAspect {
23.     private static Logger logger =
    LoggerFactory.getLogger(LoggingAspect.class);
24.
25.     @AfterThrowing("execution(*
    com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
26.     public void logAfterThrowingAdvice(JoinPoint
    joinPoint) {
27.         logger.info("In After throwing Advice,
    Joinpoint signature :{}", joinPoint.getSignature());
28.
29.     }
30.
31.     @AfterThrowing(pointcut = "execution(*
    com.infy.service.CustomerServiceImpl.fetchCustomer(..))",
    throwing = "exception")
32.     public void logAfterThrowingAdviceDetails(JoinPoint
    joinPoint, Exception exception) {
33.         logger.info("In After throwing Advice,
    Joinpoint signature :{}", joinPoint.getSignature());
34.
  logger.error(exception.getMessage(),exception);
35.
36.     }
37.
38.     @After("execution(*
    com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
39.     public void logAfterAdvice(JoinPoint joinPoint) {
40.         logger.info("In After Advice, Joinpoint
    signature :{}", joinPoint.getSignature());
41.         long time = System.currentTimeMillis();
42.         String date =
    DateFormat.getDateTimeInstance().format(time);
43.         logger.info("Report generated at time {}",
    date);
44.     }
```

```java
45.
46.        @Before("execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
47.        public void logBeforeAdvice(JoinPoint joinPoint) {
48.                // Log Joinpoint signature details
49.                logger.info("In Before Advice, Joinpoint
   signature :{}", joinPoint.getSignature());
50.                long time = System.currentTimeMillis();
51.                String date =
   DateFormat.getDateTimeInstance().format(time);
52.                logger.info("Report generated at time:{}",
   date);
53.
54.        }
55.
56.        @AfterReturning(pointcut = "execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
57.        public void logAfterReturningAdvice(JoinPoint
   joinPoint) {
58.                logger.info("In AfterReturning Advice,
   Joinpoint signature :{}", joinPoint.getSignature());
59.        }
60.
61.        @AfterReturning(pointcut = "execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))",
   returning = "result")
62.        public void logAfterReturningDetails(JoinPoint
   joinPoint, List<CustomerDTO> result) {
63.                logger.info("In AfterReturning Advice with
   return value, Joinpoint signature :{}",
   joinPoint.getSignature());
64.                System.out.println(result);
65.                long time = System.currentTimeMillis();
66.                String date =
   DateFormat.getDateTimeInstance().format(time);
67.                logger.info("Report generated at time:{}",
   date);
68.        }
69.
70.        @Around("execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
71.        public Object aroundAdvice(ProceedingJoinPoint
   joinPoint) throws Throwable {
72.                System.out.println("Before proceeding part of
   the Around advice.");
73.                 Object cust =  joinPoint.proceed();
74.                System.out.println("After proceeding part of
   the Around advice.");
```

```
75.              return cust;
76.         }
77.
78. }

79.
```

CustomerService.java

```
1. package com.infy.service;
2.
3. import java.util.List;
4.
5. import com.infy.dto.CustomerDTO;
6.
7. public interface CustomerService {
8.   public String createCustomer(CustomerDTO customerDTO);
9.
10.       public List<CustomerDTO> fetchCustomer();
11.
12.       public String updateCustomer(long phoneNumber,
   CustomerDTO customerDTO);
13.
14.       public String deleteCustomer(long phoneNumber);
15.
16. }

17.
```

CustomerServiceImpl.java

```
1. package com.infy.service;
2.
3. import java.util.List;
4.
5. import
   org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.stereotype.Service;
7.
8. import com.infy.dto.CustomerDTO;
9. import com.infy.repository.CustomerRepository;
10.
11. @Service("customerService")
```

```
12. public class CustomerServiceImpl implements
    CustomerService {
13.
14.        @Autowired
15.        private CustomerRepository customerRepository;
16.
17.        public String createCustomer(CustomerDTO
    customerDTO) {
18.
      customerRepository.createCustomer(customerDTO);
19.
20.                return "Customer with " +
    customerDTO.getPhoneNo() + " added successfully";
21.        }
22.
23.        public List<CustomerDTO> fetchCustomer() {
24.                // uncomment the below line to see the
    AfterThrowing advice
25.                // int b=10/0;
26.                return customerRepository.fetchCustomer();
27.        }
28.
29.        public String updateCustomer(long phoneNumber,
    CustomerDTO customerDTO) {
30.
31.                return
    customerRepository.updateCustomer(phoneNumber,
    customerDTO);
32.        }
33.
34.        public String deleteCustomer(long phoneNumber) {
35.
36.                return
    customerRepository.deleteCustomer(phoneNumber);
37.        }
38. }

39.
```

CustomerRepository.java

```
1. package com.infy.repository;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
```

```java
6.  import javax.annotation.PostConstruct;
7.
8.  import org.springframework.stereotype.Repository;
9.  import com.infy.dto.CustomerDTO;
10.
11.  @Repository
12.  public class CustomerRepository {
13.        List<CustomerDTO> customers = null;
14.
15.        @PostConstruct
16.        public void initializer() {
17.                CustomerDTO customerDTO = new CustomerDTO();
18.                customerDTO.setAddress("Chennai");
19.                customerDTO.setName("Jack");
20.                customerDTO.setEmail("Jack@infy.com");
21.                customerDTO.setPhoneNo(99512122221);
22.                customers = new ArrayList<>();
23.                customers.add(customerDTO);
24.
25.        }
26.
27.        // adds the received customer object to customers
  list
28.        public void createCustomer(CustomerDTO customerDTO)
  {
29.                customers.add(customerDTO);
30.        }
31.
32.        // returns a list of customers
33.        public List<CustomerDTO> fetchCustomer() {
34.
35.                return customers;
36.        }
37.
38.        // deletes customer
39.        public String deleteCustomer(long phoneNumber) {
40.                String response = "Customer of:" +
  phoneNumber + "\t does not exist";
41.                for (CustomerDTO customer : customers) {
42.                        if (customer.getPhoneNo() ==
  phoneNumber) {
43.                                customers.remove(customer);
44.                                response = customer.getName() +
  "of phoneNumber" + customer.getPhoneNo()
45.                                                + "\t got deleted
  successfully";
46.                                break;
47.                        }
```

```
48.                }
49.            return response;
50.        }
51.
52.        // updates customer
53.        public String updateCustomer(long phoneNumber,
   CustomerDTO customerDTO) {
54.            String response = "Customer of:" +
   phoneNumber + "\t does not exist";
55.            for (CustomerDTO customer : customers) {
56.                if (customer.getPhoneNo() ==
   phoneNumber) {
57.
58.                    if (customerDTO.getName() !=
   null)
59.
   customer.setName(customerDTO.getName());
60.                    if (customerDTO.getAddress() !=
   null)
61.
   customer.setAddress(customerDTO.getAddress());
62.
63.
   customers.set(customers.indexOf(customer), customer);
64.                    response = "Customer of
   phoneNumber" + customer.getPhoneNo() + "\t got updated
   successfully";
65.                    break;
66.                }
67.            }
68.            return response;
69.        }
70.
71. }
72.
```

Demo9Application.java

```
1.    package com.infy;
2.
3.    import
   org.springframework.boot.SpringApplication;
```

```
4.      import
org.springframework.boot.autoconfigure.SpringBootApplic
ation;
5.      import
org.springframework.context.support.AbstractApplication
Context;
6.      import com.infy.service.CustomerServiceImpl;
7.
8.      @SpringBootApplication
9.      public class Demo9Application {
10.
11.         public static void main(String[] args) {
12.
13.             CustomerServiceImpl service = null;
14.             AbstractApplicationContext context =
(AbstractApplicationContext)
SpringApplication.run(Demo9Application.class,
15.                         args);
16.             service = (CustomerServiceImpl)
context.getBean("customerService");
17.             service.fetchCustomer();
18.             context.close();
19.         }
20.
21.     }
```

Summary :AOP

This module helped us in learning the following:

- What is AOP?
- Different types of Advices used in AOP
- How to implement AOP in Spring?

Best Practices : Spring Boot Application

Let us discuss the best practices which need to be followed as part of the Quality and Security for the Spring application and Spring with Spring Boot applications. These practices, when applied during designing and developing a Spring/Spring Boot application, yields better performance.

Best Practices:

1. To create a new spring boot project prefer to use Spring Initializr

2. While creating the Spring boot projects must follow the Standard Project Structure

3. Use @Autowired annotation before a constructor.

4. Use constructor injection for mandatory dependencies and Setter injection for optional dependencies in Spring /Spring boot applications

5. Inside the domain class avoid using the stereotype annotations for the automatic creation of Spring bean.

6. To create a stateless bean use the singleton scope and for a stateful bean choose the prototype scope.

Let us understand the reason behind these recommendations and their implications.

There are three different ways to create a Spring Boot project. They are:

- Using Spring Initializr
- Using the Spring Tool Suite (STS)
- Using Spring Boot CLI

But the recommended and simplest way to create a Spring Boot application is the Spring Boot Initializr as it has a very good UI to download a production-ready project. And the same project can be directly imported into the STS/Eclipse.

Note: The above screen keeps changing depending on updates from Pivotal and changes in the Spring Boot version.
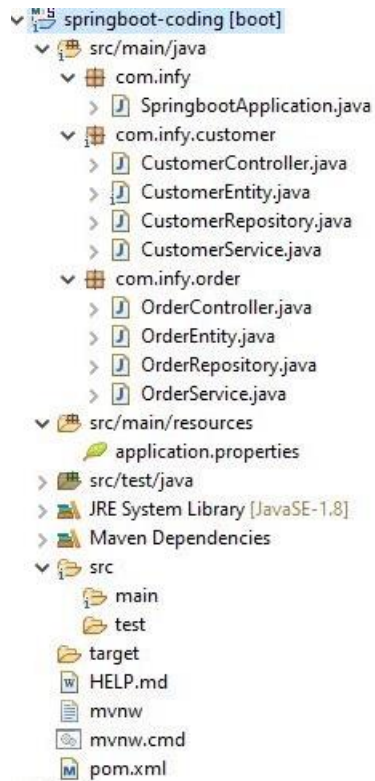
There are two recommended ways to create a spring boot project structure for Spring boot applications, which will help developers to maintain a standard coding structure.

Don't use the "default" Package:

When a developer doesn't include a package declaration for a class, it will be in the "default package". So the usage of the "default package" should be avoided as it may cause problems for Spring Boot applications that generally use the annotations like @ComponentScan, or @SpringBootApplication. So, during the creation of the classes, the developer should follow Java's package naming conventions. For example, com.infy.client, com.infy.service, com.infy.controller etc.

There are 2 approaches that can be followed to create a standard project structure in Spring boot applications.

First approach: The first approach shows a layout which generally recommended by the Spring Boot team. In this approach we can see that all the related classes for the customer have grouped in the **"com.infy.cutomer"** package and all the related classes for the order have grouped in the **"com.infy.order"** package

```
v [M:S] springboot-coding [boot]
  v (⊞ src/main/java
    v ⊞ com.infy
      > [J] SpringbootApplication.java
    v ⊞ com.infy.customer
      > [J] CustomerController.java
      > [J] CustomerEntity.java
      > [J] CustomerRepository.java
      > [J] CustomerService.java
    v ⊞ com.infy.order
      > [J] OrderController.java
      > [J] OrderEntity.java
      > [J] OrderRepository.java
      > [J] OrderService.java
  v (⊞ src/main/resources
        [🍃] application.properties
  > (⊞ src/test/java
  > [▨] JRE System Library [JavaSE-1.8]
  > [▨] Maven Dependencies
  v (👉 src
        (👉 main
        (👉 test
    (👉 target
    [w] HELP.md
    [▤] mvnw
    [👁] mvnw.cmd
    [M] pom.xml
```

Second approach: However the above structure works well but developers prefer to use the following structure.

In this approach, we can see that all the service classes related to customer and Order are grouped in the **"com.infy.service"** package. Similar to that we can see we grouped the repository, controller, and entity classes.
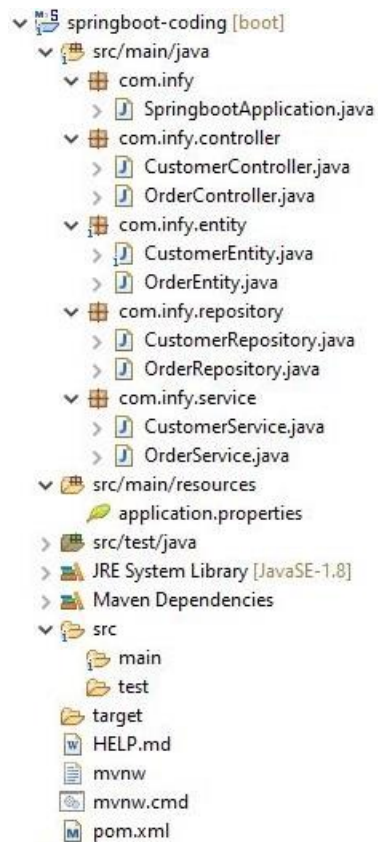
Best Practices : Spring Application

- **Inside the domain class, try to avoid using the stereotype annotations for the automatic creation of Spring bean.**

```
1. @Component
2.
3. public class Employee {
4.
5.  // Methods and variables
6.
7. }
```

Avoid creating beans for the domain class like the above.

- **Avoid scanning unnecessary classes to discover Beans. Specify only the required class for scanning.**

Suppose that if we need to discover beans declared inside the service package. Let us write the code for that.

```
1. @Configuration
2.
3. @ComponentScan("com.infosys")
4.
5. public class AppConfig {
6.

7. }
```

In the above configuration class, we mentioned scanning the entire package com.infosys. But our actual requirement is only needed to scan the service packages. We can avoid this unnecessary scanning by replacing the code as below.

```
1. @Configuration
2.
3. @ComponentScan("com.infosys.service")
4.
5. public class AppConfig {
6.
7. }
```

- **Java doesn't allow annotations placed on interfaces to be inherited by the implemented class so make sure that we place the spring annotations only on class, fields, or methods.**
- **As a good practice place @Autowired annotation before a constructor.**

We know that there are three places we can place @Autowired annotation in Spring on fields, on setter method, and on a constructor. The classes using field injection are difficult to maintain and it is very difficult to test. The classes using setter injection will make testing easy, but this has some disadvantages like it will violate encapsulation. Spring recommends that use @Autowired on constructors that are easiest to test, and the dependencies can be immutable.

- **In the case of AOP as a best practice store all the Pointcuts in a common class which will help in maintaining the pointcuts in one place.**

```
1. public class CommonPointConfig {
2.
3.   @Pointcut("execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ")
4.         public void logAfterAdvice(JoinPoint
   joinPoint){}
5.         @Pointcut("execution(*
   com.infy.service.CustomerServiceImpl.fetchCustomer(..))
   ")
6.         public void logBeforeAdvice(){}
7. }

8.
```

The above common definition can be used when defining pointcuts in other aspects.

```
1. @Around("com.infy.service.CustomerServiceImpl.fetchCust
   omer.aspect.CommonPointConfig.logBeforeAdvice()")

2.
```

- **While defining the scope of the beans choose wisely.**

If we want to create a stateless bean then singleton scope is the best choice. In case if you need a stateful bean then choose prototype scope.

- **When to choose Constructor-based DI and setter-based DI in Spring /Spring boot applications**

A Spring application developer can mix the Constructor injection and Setter injection in the same application but it's a good practice to use constructor injection for mandatory dependencies and Setter injection for optional dependencies.

Summary :Best Practices

This module helped us in learning the following:

- The Best Practices need to follow in the Spring application.
- The Best Practices needs to follow in Spring Boot.

Appendix A: AOP
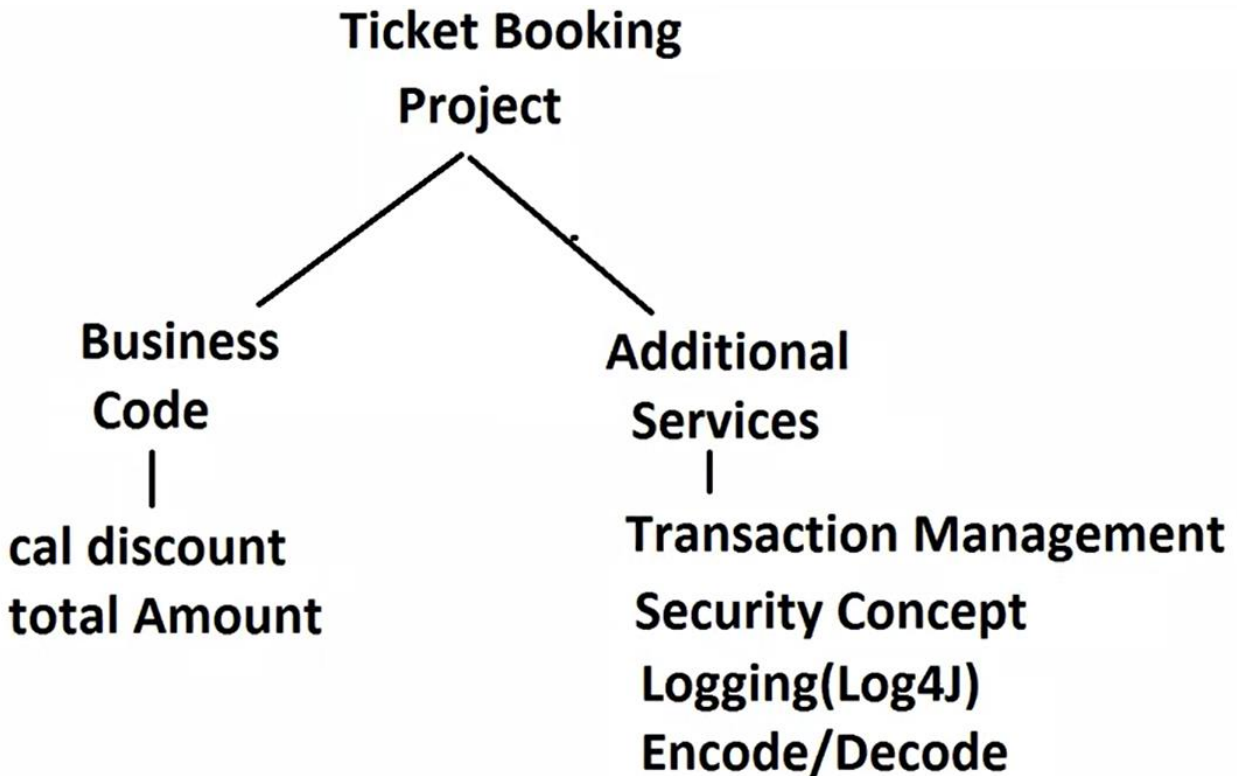

```
void bookTocket(Ticket  t) {
    beginTx();

    //cal disc
    //cal total..
      dao.save(t);
    commitTx();
}
```

Concerns

void bookTocket(Ticket t) {
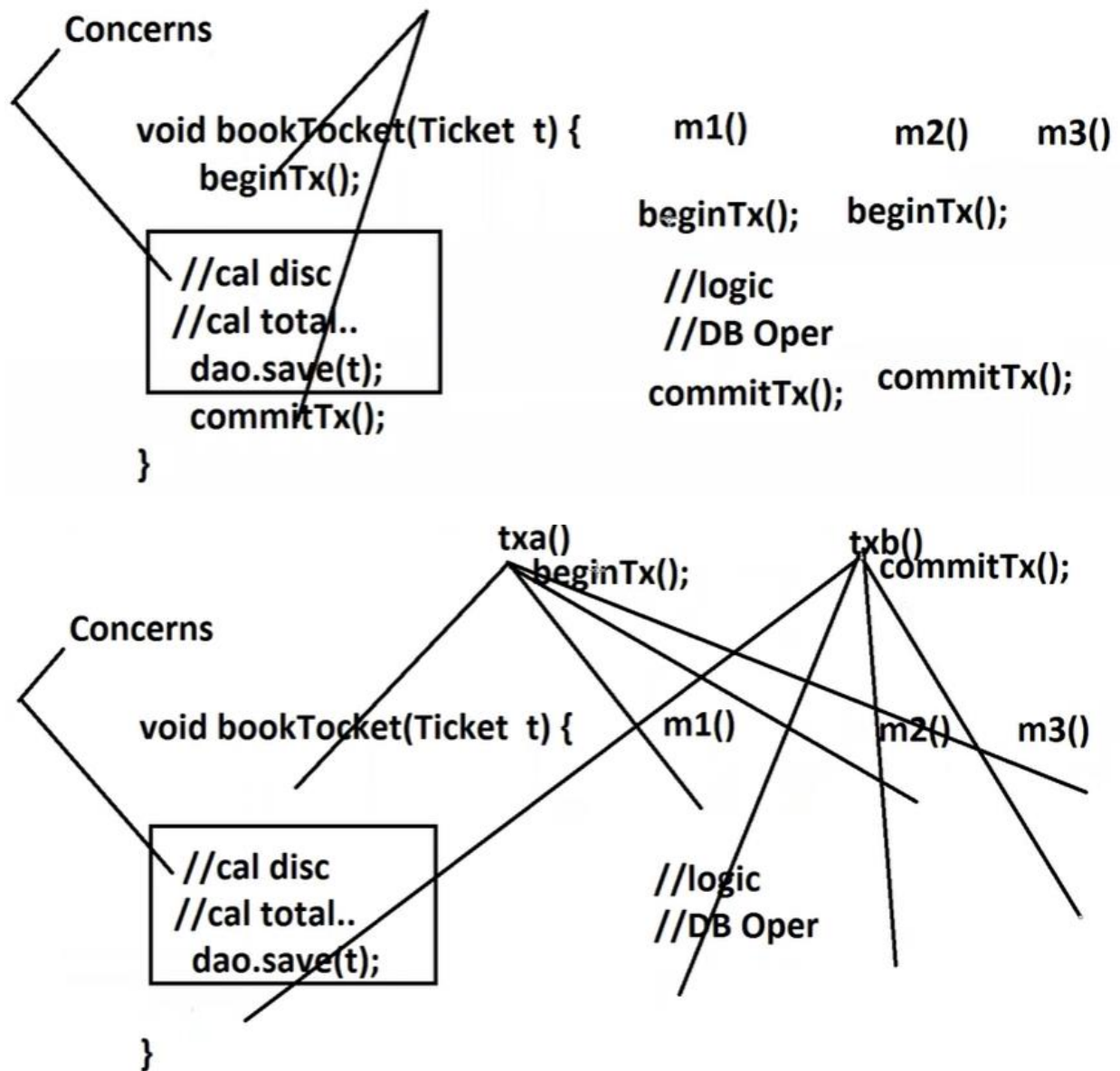   beginTx();

        //cal disc
        //cal total..
        dao.save(t);
        commitTx();
}

m1()   m2()   m3()

        beginTx();   beginTx();

        //logic
        //DB Oper

        commitTx();   commitTx();

Concerns

        txa()   txb()
        beginTx();   commitTx();

void bookTocket(Ticket t) {   m1()   m2()   m3()

        //cal disc
        //cal total..
        dao.save(t);

        //logic
        //DB Oper

}

```
----------------
AOP : Aspect Oriented Programming
```

```
*) Cross-Cutting Concern: Move Additional Services of a Project into other classes
   [Services class/Aspect classes] and bind when and wher they are required.
```

Business class   **Additional Services class**

class TicketService {
                              selecting b.method

        bookTicket() {..}   class TxService {   Advices
b.methods   ✕   beginTx() { ...}

        printTicket() {...}   commitTx() { ...}

}   }

                              Target

Business class   **Additional Services class**

class TicketService {   **(Pointcut)**   Aspect
                              selecting b.method   class TxService {   Advices
        bookTicket() {..}   beginTx() { ...}
b.methods   ✕   **Joinpoint**
        printTicket() {...}   commitTx() { ...}

}   }

───

JDK Dynamic Proxy

        bookTicket() {..}   **Joinpoint**   beginTx() { ...}
b.methods   ✕
        printTicket() {...}   commitTx() { ...}

}   }

───

JDK Dynamic Proxy

        Weaving

        class __Proxy {
            b.method+advices

        }

Proxy Object

1) Aspect   : Aspect is a class, that provides additional services to project.
                Transaction Management, Logging, Security, Encode and Decode..etc.

2) Advice   : It is a method inside Aspect class. [It is actual implementation of Aspect]

3) Pointcut : It is expression, it will select b.method which needs advices.
                It can never specify what advices.

4) JoinPoint : It is a combination of Advices + Pointcut. In simple connecting b.methods
                with required advices.

5) Target   : Pure Business class object.

6) Weaving : It is a process of mixing b.class methods and their connected advices

7) Proxy   : Final Output (class/object) is called as Proxy that contains both
                logic connected.

Implementations
a) Spring AOP using XML Based Configuration [Legacy Style]
b) Spring AOP using AspectJ [Annotations]

2) Advice   : It is a method inside Aspect class. [It is actual implementation of Aspect]

Types of Advices(5)
Before Advice   : Executing Advice before calling b.method
After Advice   : Executing Advice After b.method finished
Around Advice   : Advice code made into 2 sections/parts, 1st part executed
                             before advice and then b.method, later 2nd part of advice.

After Returning Advice : After executing b.method, only on sucess execute advice.
After Throwing Advice : After executing b.method, if b.method is throwing any
                             exception then execute advice.

 Pointcut : It is expression, it will select b.method which needs advices.
            It can never specify what advices.

expression :   AS RT PACK.CLASS.METHOD(PARAM)

```java
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;


@Aspect
public class TxService {

    @Pointcut("execution(public *  com.app.ProductDao.*(..))")
    public void p1() {}

    //@Before("")
    //@After("")
    //@Around("")
    //@AfterReturning("")
    //@AfterThrowing("")
    public void beginTx() {
        //...
    }
}
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```
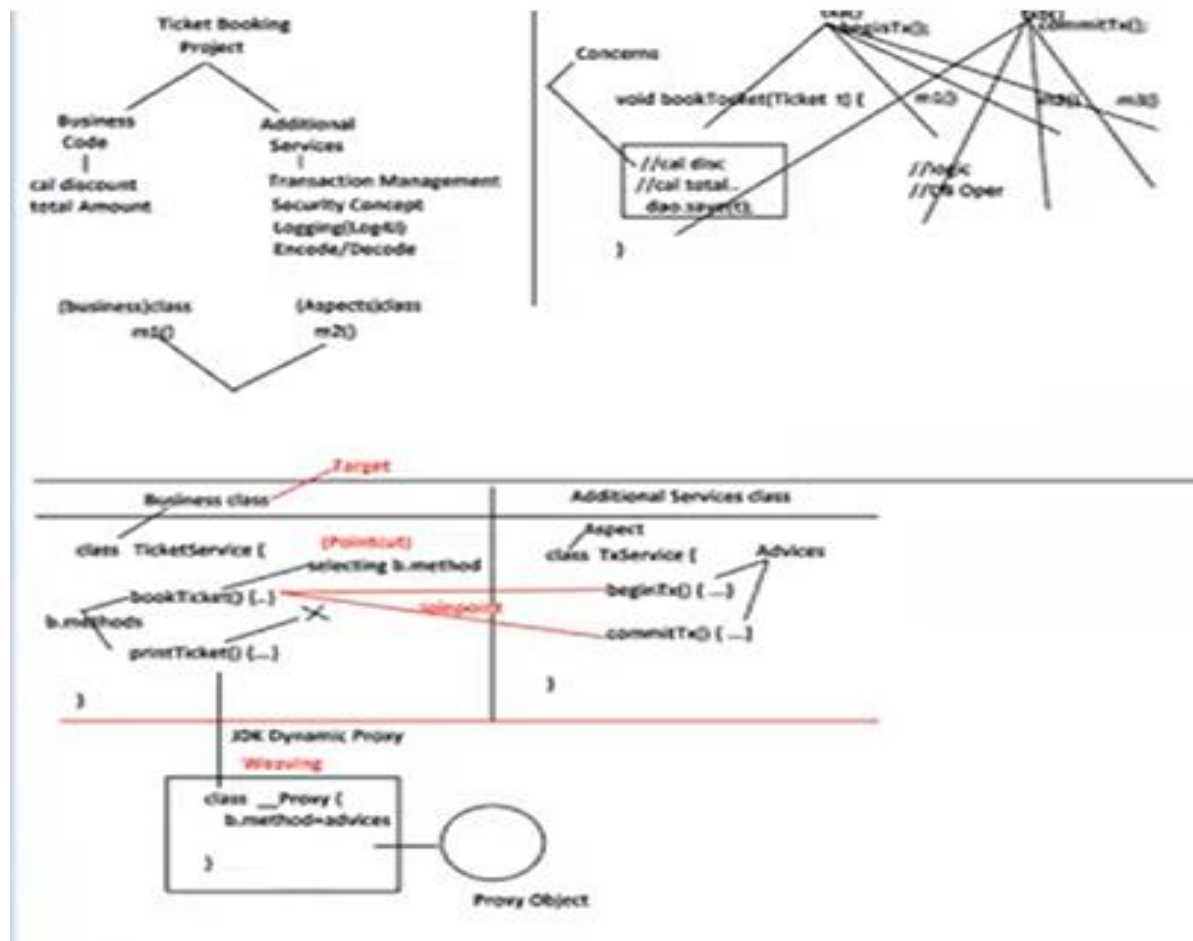
Annotations:
@Aspect
@Before
@After
@Around
@AfterReturning
@AfterThrowing
@Pointcut

b.method	advices	Type
m1()- Hello	a1() - Welcome	— Before
m2()- Bye	a2() - NIT	— After
m3()- OK	a3() - $\frac{Test}{Done}$	— Around
m4()- NullPointerException	a4() - Nice	—— AfterReturning
m5()- Success	a5() - Fine	—— AfterThrowing

```
Types of Advices(5)
Before Advice            : Executing Advice before calling b.method

        execution order :  adviceMethod()  -- 1st
                           b.method()       -- 2nd


After Advice             : Executing Advice After b.method finished

        execution order :  b.method()       -- 1st
                           adviceMethod()   -- 2nd

 Around Advice           : Advice code made into 2 sections/parts, 1st part executed
                           before advice and then b.method, later 2nd part of advice.

        execution order : adviceMethod()  -- 1st-Part
                          b.method()
                          adviceMethod()  -- 2nd-Part

 After Returning Advice : After executing b.method, only on sucess execute advice.

        execution order :  b.method()      -- 1st
                (Is B.method executed successfully, no exception) then call
                        adviceMethod()  -- 2nd

 After Throwing Advice  : After executing b.method, if b.method is throwing any
                          exception then execute advice.

        execution order :  b.method()      -- 1st
                (Is B.method executed not successfully, exception occured) then call
                        adviceMethod()  -- 2nd
```
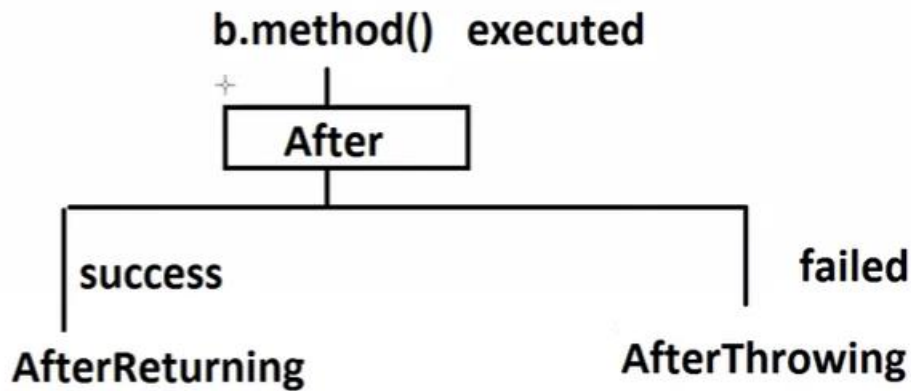
b.method() executed

```
         +
          |
    ┌─────────────┐
    │    After    │
    └─────────────┘
    │             │
 success        failed
AfterReturning   AfterThrowing
```

Q) What is the difference between After, After Returning, After Throwing Advices?
A) After advices is executed next to b.method either success or fail.
   After Returning advice is executed only on succssful execution of b.method
   After Throwing advice is executed only on Failure(Exception) execution of b.method

Pointcut Syntax:-
 Specifier ReturnType   Package.ClassName.MethodName(ParameterTypes)

Note : allowed symbols in Pointcut expression: *(star), .(dot)

-------Examples--
#1) public int in.nit.dao.EmployeeDao.saveEmployee(Employee)

=> saveEmployee() method having Parameter 'Employee' with return type 'int'
   of type public defined in class EmployeeDao (in.nit.dao)
   is selected to connecte with advice.

#2) public * in.nit.dao.EmployeeDao.*(|)

  => Zero Parametes only (no parameter)
  => Any method name/ Method inside EmployeeDao class
  => Any return type

#3) public * in.nit.dao.EmployeeDao.*(..)

  => Zero or more params (of any type) [Any Parameter is fine]

#4) public * in.nit.dao.*.*(..)   I

 => all classes which ar inside in.nit.dao package and thier methods

```
Aspect (class)
Advice (method)
 Before     advice()-->b.method()

 After      b.method()-->success/fail--> advice()

 AfterReturning     b.method()-->only on success--> advice()
 AfterThrowing      b.method()-->only on fail--> advice()

   Around
Pointcut - expression-selecting b.method


===================================================================

Before  execution of saveEmployee() method plz call beginTx() method

 Aspect (class)
 Advice (method)
  Before     advice()-->b.method()

  After      b.method()-->success/fail--> advice()

  AfterReturning     b.method()-->only on success--> advice()
  AfterThrowing      b.method()-->only on fail--> advice()

    Around -- advice-part1() --->b.method()--->advice-part-2()


 Pointcut - expression-selecting b.method
```

```java
@Aspect
@Component
public class TxService {

    // pointcut : AS RT PACK.CLS.MN(PARAM)
    @Pointcut("execution(public void in.nareshit.raghu.dao.EmployeeDao.saveEmployee(
    public void p1() {}

    // advice
    @Before("p1()") //joinpoint
    public void beginTx() {
        System.out.println("Tx Started!!");
    }

    @After("p1()")
    public void sendReport() {
        System.out.println("Report sent!");
    }

}
```

```java
package in.nareshit.raghu.dao;

import org.springframework.stereotype.Component;

@Component
public class EmployeeDao  {

    public void saveEmployee() {
        System.out.println("FROM SAVE EMPLOYEE");

    }
}
```
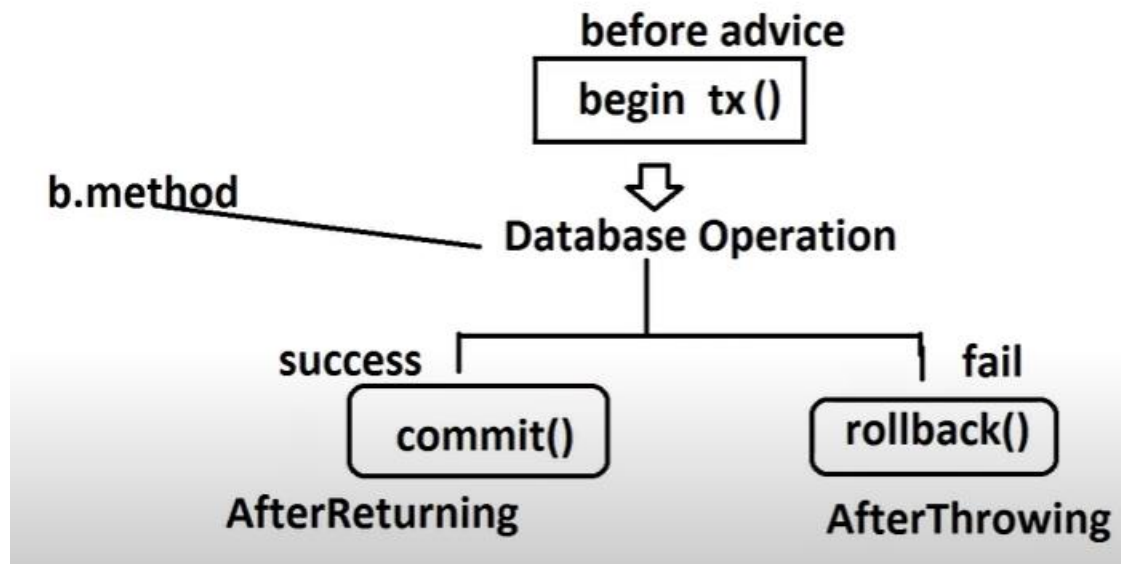
```java
import org.springframework.boot.SpringApplication;

@SpringBootApplication
@EnableAspectJAutoProxy
public class SpringBoot2AopExApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBoot2AopExApplication.class, args);
    }

}
```

before advice

begin tx ()

b.method

Database Operation

success

commit()

AfterReturning

fail

rollback()

AfterThrowing

boilerplate code or 'boilerplate'

part of code that can be reused over and over again without modification