API AND MICROSERVICES
(Job Oriented Course)

Course Outcomes:

At the end of this course, the student will be able to

- Develop a Spring Data JPA application with Spring Boot
- Implement CRUD operations using Spring Data JPA
- Implement pagination and sorting mechanism using Spring Data JPA
- Implement query methods for querying the database using Spring Data JPA
- Implement a custom repository to customize a querying mechanism using Spring Data JPA
- Understand update operation using query approaches in Spring Data JPA
- Implement Spring Transaction using Spring Data JPA
- Develop RESTful endpoints using Spring REST Processing URI parameters
- Write RESTful services using Spring REST that consumes and produces data in different formats
- Handle exceptions and errors in Spring REST endpoints
- Write Spring based REST clients to consume RESTful services programmatically
- Create secure RESTful endpoints using Spring Security Document and version the Spring REST endpoints Implement CORS in a Spring REST application

UNIT I:

Spring 5 Basics : Why Spring, What is Spring Framework, Spring Framework - Modules, Configuring IoC container using Java-based configuration, Introduction To Dependency Injection, Constructor Injection, Setter Injection, What is AutoScanning

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

UNIT III:

Spring Data JPA with Boot: Limitations of JDBC API, Why Spring Data JPA, Spring Data JPA with Spring Boot, Spring Data JPA Configuration, Pagination and Sorting, Query Approaches, Named Queries and Query, Why Spring Transaction, Spring Declarative Transaction, Update Operation in Spring Data JPA, Custom Repository Implementation, Best Practices - Spring Data JPA

UNIT IV:

Web Services: Why Web services, SOA - Service Oriented Architecture, What are Web Services, Types of Web Services, SOAP based Web Services, RESTful Web Services, How to create RESTful Services

UNIT V:

Spring REST: Spring REST - An Introduction, Creating a Spring REST Controller, @RequestBody and ResponseEntity, Parameter Injection, Usage of @PathVariable, @RequestParam and @MatrixVariable, Exception Handling, Data Validation, Creating a REST Client, Versioning a Spring REST endpoint, Enabling CORS in Spring REST, Securing Spring REST endpoints

Hardware and software configuration

- 4 or 8 GB RAM/126 GB ROM
- Swagger tool suite(opensource)
- OpenJDK 17 or Java 11,Maven 3.2 or above and MySQL 8.0 or above,Spring Tool suite, Postman

Text Books:

1. Spring in action, 5th Edition, Author: Craig Walls, Ryan Breidenbach, Manning books

Web Links [Courses mapped to Infosys Springboard platform]:

Infosys Springboard courses:

1. https://infyspringboard.onwingspan.com/en/app/toc/lex_auth_01296689056211763272_shared/overview [Spring 5 Basics with Spring Boot]
2. https://infyspringboard.onwingspan.com/en/app/toc/lex_4313461831752789500_shared/overview [Spring Data JPA with Boot]
3. https://infyspringboard.onwingspan.com/en/app/toc/lex_auth_01273190096390553619 0_shared/overview [Spring REST]

API & Micro Services

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

Introduction to Autowiring

Spring

1. Introduction to Spring Framework
2. Spring Inversion of Control (IoC)
3. Dependency Injection (DI)
4. Autowiring
   - 4.1 Understand Autowiring in Spring
   - 4.2 Implement Autowiring using Spring XML configuration
5. Bean Scope
6. Spring Annotation and Java based configuration
7. Spring AOP

Spring

Introduction to Autowiring

- It is not required to specify all the properties values while defining bean in the configuration. For non-primitive properties we can use the Autowiring feature.
- If Autowiring is enabled,
  - Spring container automatically initializes the dependencies of that bean.
- There is no need to provide explicit <property > or <constructor-arg> with ref tags in the bean definition for object dependencies.
- Autowiring cannot be used to inject primitive and string values.

Introduction to Autowiring

Spring

Spring automatically inject the bean dependency. The autowiring has four modes.

- no (no autowiring)
- byName
- byType
- constructor

In XML configuration, autowire mode is specified in the bean definition using the **autowire** attribute.

Introduction to Autowiring

Spring

Autowiring Modes

Mode	Description
byName	• Bean dependency is autowired based on the property name • If the matching bean does not exist in the container then bean will remain unwired • It internally uses setter injection
byType	• Autowiring the bean dependency based on the property type • Properties for which there is no matching bean will remain unwired. • Spring throws an exception if there is more than one bean of the same type exists in the container • It internally uses setter injection
Constructor	• It is the same as autowiring byType but through constructor arguments. • Spring autowire the dependency based on constructor argument type through constructor injection. • Spring throws an exception if there is more than one bean of same type exists in the container
No	Default mode which means no autowiring

Introduction to Autowiring

Autowiring using byName mode

Spring

Let us understand how the bean dependencies are autowired using "**byName**" mode in Spring.

In byName mode, Spring looks for a bean in the container with id same as property name to autowire the dependency.

In the below given code, ReportService class is dependent on ReportGenerator bean.

```
1. package com.infosys.demo;
2. public class ReportService {
3.         private ReportGenerator master;
4.         private int recordsPerPage;
5.     ---------------------------
6. }
7.
```

Introduction to Autowiring

Autowiring using byName mode

Spring

For autowiring byName, following details are required in the configuration.

- Define bean of ReportGenerator with id same as property name (Here it is "master" in the above class)
- Mention an attribute autowire value "byName" in the reportService bean definition as shown below.

```
1. <bean id="reportService" class="com.infosys.demo.ReportService"autowire="byName">
2.                 <property name="recordsPerPage" value="500" />
3. </bean>
4.
5. <bean id="master" class="com.infosys.demo.PDFReportGenerator" />
6.
```

In byName mode, it is mandated to have bean id same as the property name and also the respective setter methods as Spring inject the bean dependency internally using setter injection.

Introduction to Autowiring

Spring

Demo : Autowiring byName

Highlights:

- Objective:To apply Autowiring value "byName" in XML configuration
- Required Jars: Same as Demo1

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Autowiring

Spring

Demo : Autowiring byName

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}
```

Introduction to Autowiring

Demo : Autowiring byName

Spring

ReportService.java --> Service class

```java
package com.infosys.demo;
public class ReportService {
    private ReportGenerator master;
    private int recordsPerPage;
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        System.out.println("Setter Method of RecordsPerPage Property");
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    public void setMaster(ReportGenerator master) {
        System.out.println("Setter Method of master Property");
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Introduction to Autowiring

Spring

Demo : Autowiring byName

config.xml --> Spring configuration in XML

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="reportService" class="com.infosys.demo.ReportService" autowire="byName">
      <property name="recordsPerPage" value="500" />
   </bean>

   <bean id="htmlReportGenetator" class="com.infosys.demo.HTMLReportGenerator" />

   <bean id="master" class="com.infosys.demo.PDFReportGenerator" />

</beans>
```

Introduction to Autowiring

Demo : Autowiring byName

Spring

Client.java --> Client Code

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6.  public class Client {
7.
8.      public static void main(String[] args) {
9.          ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
10.         ReportService srv = (ReportService)context.getBean("reportService");
11.         srv.generateReport();
12.
13.     }
14.
15. }
```

OUTPUT

```
1.  Setter Method of RecordsPerPage Property
2.  Setter Method of master Property
3.  Generated PDF Report with 500 records
4.
```

Introduction to Autowiring

Autowiring using byType mode

Spring

Let us understand how the bean dependencies are autowired using "**byType**" mode in Spring.

In byType mode, Spring looks for the bean in the container based on type of bean to autowire the dependency.

In the below given code, ReportService class is dependent on ReportGenerator bean.

```
1. package com.infosys.demo;
2. public class ReportService {
3.         private ReportGenerator master;
4.         private int recordsPerPage;
5.         ----------------------------
6. }
7.
```

In order to use autowiring byType, we need to provide following details in the configuration.

- Define bean of ReportGenerator with any bean id
- Mention an attribute autowire value "byType" in the reportService bean definition as shown below.

Introduction to Autowiring

Autowiring using byType mode

Spring

```
<bean id="reportService" class="com.infosys.demo.ReportService" autowire="byType">
   <property name="recordsPerPage" value= "500"/>
</bean>

<bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
```

In byType mode, there is no mandate to have bean id same as property name as the dependency now is autowired based on type of property instead of the name. byType also requires the respective setter methods in the class as Spring inject the bean dependency internally using setter injection.

In the previously discussed autowire byName demo, modify autowire attribute value to "byType" in the reportService bean definition and execute the demo.

Introduction to Autowiring

Autowiring using constructor mode

Spring

In this mode, Spring does autowiring of beans internally byType similar to that of byType mode however here the dependency is injected using constructor injection instead of setter injection. Hence it is required to have parameterized constructor in the respective class.

In the below-given code, ReportService class is dependent on ReportGenerator bean.

```
package com.infosys.demo;
public class ReportService {
        private ReportGenerator master;
        private int recordsPerPage;
        public ReportService(ReportGenerator master, int recordsPerPage) {
                this.master = master;
                this.recordsPerPage = recordsPerPage;
        }
        public ReportService() {

        }
        ---------------------------------
        ---------------------------------
}
```

Introduction to Autowiring

Autowiring using constructor mode

Spring

For using constructor autowiring, we need to specify the following details through configuration:

- Define bean of ReportGenerator with any bean id
- Mention an attribute autowire with value "constructor" in the reportService bean definition as shown below.

```
<bean id="reportService" class="com.infosys.demo.ReportService" autowire="constructor">
    <constructor-arg name="recordsPerPage" value= "500"/>
  </bean>

  <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator"/>
```

In constructor mode of autowiring, again there is no mandate to have bean id same as property name as the dependency now is autowired based on type of property instead of the name. However it is mandatory to have parametarized constructor in the ReportService class because the dependency is now injected through the constructor.

Introduction to Autowiring

Spring

Demo : Autowiring by constructor mode

Highlights:

- Objective: To learn how to apply Autowiring by constructor mode.
- Required Jars: Same as Demo1

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Autowiring

Spring

Demo : Autowiring by constructor mode

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}
```

AUK

Introduction to Autowiring

Demo : Autowiring by constructor mode

ReportService.java --> Service class

```java
package com.infosys.demo;
public class ReportService {
    private ReportGenerator master;
    private int recordsPerPage;
    public ReportService(ReportGenerator master, int recordsPerPage) {
        System.out.println("Parameterized Constructor");
        this.master = master;
        this.recordsPerPage = recordsPerPage;
    }
    public ReportService() {
        System.out.println("Default Constructor");
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        System.out.print("Setter Method of recordsPerPage property");
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    public void setMaster(ReportGenerator master) {
        System.out.print("Setter Method of master property");
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Introduction to Autowiring

Spring

Demo : Autowiring by constructor mode

config.xml--> Spring configuration in XML

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <beans xmlns="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.      xsi:schemaLocation="http://www.springframework.org/schema/beans
6.      http://www.springframework.org/schema/beans/spring-beans.xsd">
7.
8.      <bean id="reportService" class="com.infosys.demo.ReportService" autowire="constructor">
9.          <constructor-arg name="recordsPerPage" value="150" />
10.     </bean>
11.     <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
12. </beans>
13.
```

Introduction to Autowiring

Spring

Demo : Autowiring by constructor mode

Client.java --> Client Code

```java
package com.infosys.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
        ReportService srv = (ReportService)context.getBean("reportService");
        srv.generateReport();
    }
}
```

OUTPUT

```
Parameterized Constructor
Generated PDF Report with 150 records
```

Introduction to Autowiring

Autowiring using no mode

Spring

Another mode of autowiring is "no", which means no autowiring. Dependency is explicitly wired using <property name="propertyName" ref="beanId"> in setter injection or <constructor-arg ref="beanId"> in constructor injection configuration in the configuration.

In the below-given code, ReportService class is dependent on ReportGenerator bean.

```
1.  package com.infosys.demo;
2.
3.  public class ReportService {
4.
5.          private ReportGenerator master;
6.          private int recordsPerPage;
7.
8.          ------------------------------------
9.
10. }
11.
```

Introduction to Autowiring

Autowiring using no mode

Spring

Following is the required configuration, wherein reportService bean dependent beans are explicitly autowired.

```
1. <bean id="reportService" class="com.infosys.demo.ReportService"  autowire="no">
2.                 <property name="master" ref="htmlGenerator" />
3.                 <property name="recordsPerPage" value="500" />
4. </bean>
5.
6. <bean id="htmlReportGenerator"class="com.infosys.demo.HTMLReportGenerator" />
7.
```

"no" mode of autowiring is the **default** one in case of autowiring attribute is not present explicitly in the bean definition.

Introduction to Autowiring

Autowiring using no mode

Spring

Following reportService bean definition is equivalent to the above shown reportService bean definition.

```
1. <bean id="reportService" class="com.infosys.demo.ReportService">
2.                 <property name="master" ref="htmlGenerator" />
3.                 <property name="recordsPerPage" value="500" />
4. </bean>
5.
```

In the previous demo, modify the autowire attribute value to "no" in the Spring configuration reportService bean definition, inject the required dependency explicitly and observe the result.

Introduction to Autowiring

Spring

Summary : Autowire Modes

Autowiring Mode	Description
byName	Autowiring based on bean name through setter injection
byType	Autowiring based on bean type through setter injection
constructor	Autowiring based on the bean type through parameterized constructor
no	No Autowiring, required dependency has to be injected explicitly using <property> or <constructor-arg> ref attribute in bean definition

Bean Scope

Spring

1. Introduction to Spring Framework
2. Spring Inversion of Control (IoC)
3. Dependency Injection (DI)
4. Autowiring
5. Bean Scope
   - 5.1 Understand Spring Bean Scopes
   - 5.2 Singleton scope
   - 5.3 Prototype scope
6. Spring Annotation and Java based configuration
7. Spring AOP

Bean Scope

- 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 "**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 case of "**prototype**" scope, a new bean instance is created for every bean request from the application.

Spring

Bean Scope

Bean Scope - singleton

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

In XML configuration, bean scope is specified using the "scope" attribute as shown in the below example.

Here, reportService bean is defined as **singleton** scope.

```
<bean    id="reportService"    class="com.infosys.demo.ReportService"
scope="singleton"/>
```

This is the default mode, if not specified explicitly in the bean definition. In the below example, reportService bean has a singleton scope.

```
1. <bean id="reportService" class="com.infosys.demo.ReportService"/>
2.
```

Spring

Bean Scope

Bean Scope - singleton

Consider the Report Generation application, let us define reportService bean as a singleton bean as shown below.

```
1. <bean id="reportService" class="com.infosys.demo.ReportService" scope="singleton"
   >
2.                 <property name="master" ref="htmlReportGenerator" />
3.                 <property name="recordsPerPage" value="500" />
4.         </bean>
5.
6. <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
7.
```

In the above example, both reportService and htmlReportGenerator (since attribute scope does not exist in the bean definition, by default it is a singleton) beans are in singleton scope.

Bean Scope

Bean Scope - singleton

Spring

Let us access reportService bean in the client class more than once and compare their hashCode to see whether we are getting the same bean or different bean.

```
1.  public class Client {
2.          public static void main(String[] args) {
3.                  ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
4.                  ReportService srv1 = (ReportService)context.getBean("reportService");
5.                  ReportService srv2 = (ReportService)context.getBean("reportService");
6.                  System.out.println("hash code of srv1:" + srv1.hashCode());
7.                  System.out.println("hash code of srv2:" +srv2.hashCode());
8.                  if(srv1==srv2){
9.                          System.out.println("Same instance");
10.                 }
11.                 else
12.                         System.out.println("Different instance");
13.         }
14.
15. }
16.
```

Spring

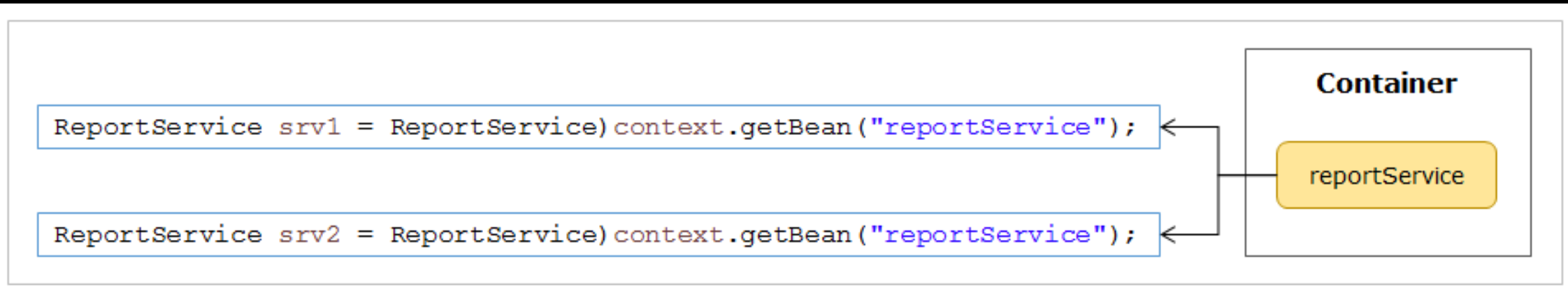
Bean Scope

Bean Scope - singleton

Execution of the above client code gives below result.

```
1. hash code of srv1:1690287238
2. hash code of srv2:1690287238
3. Same instance
```

In the above output response, you can see that hashCode of srv1 and srv2 are same indicating they both are referring to same bean instance in the container.



Bean Scope

Demo : Singleton bean scope

Highlights:

- Objective: To use a bean with 'singleton' scope
- Required Jars: Required Jars: Same as Demo1

Demosteps:

ReportService.java --> Service class

```
package com.infosys.demo;
public class ReportService {
        private ReportGenerator master;
        private int recordsPerPage;
        public int getRecordsPerPage() {
                return recordsPerPage;
        }
        public void setRecordsPerPage(int recordsPerPage) {
                this.recordsPerPage = recordsPerPage;
        }
        public ReportGenerator getMaster() {
                return master;
        }
        public void setMaster(ReportGenerator master) {
                this.master = master;
        }
        public void generateReport() {
                System.out.println(master.generateReport(recordsPerPage));
        }
}
```

Bean Scope

Demo : Singleton bean scope

Spring

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator  implements ReportGenerator{

	@Override
	public String generateReport(int recordsPerPage) {

		return "Generated HTML Report  with " + recordsPerPage + " records";
	}
}
```

Bean Scope

Demo : Singleton bean scope

Spring

Config.xml --> Spring configuration in XML

```
1.  ?xml version="1.0" encoding="UTF-8"?>
2.
3.  <beans xmlns="http://www.springframework.org/schema/beans"
4.          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.          xsi:schemaLocation="http://www.springframework.org/schema/beans
6.      http://www.springframework.org/schema/beans/spring-beans.xsd">
7.
8.          <bean id="reportService" class="com.infosys.demo.ReportService"  scope="singleton">
9.                  <property name="master" ref="htmlReportGenerator" />
10.                 <property name="recordsPerPage" value="500" />
11.         </bean>
12.         <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
13.         <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
14. </beans>
15.
```

Bean Scope

Demo : Singleton bean scope

Spring

Client.java --> Client Code

```
package com.infosys.demo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Client {
        public static void main(String[] args) {
                ApplicationContext context = new ClassPathXmlApplicationContext("Config.xml");
                ReportService srv1 = (ReportService)context.getBean("reportService");
                ReportService srv2 = (ReportService)context.getBean("reportService");
                System.out.println(srv1.hashCode());
                System.out.println(srv2.hashCode());
                if(srv1==srv2){
                        System.out.println("Same instance");
                }
                else
                        System.out.println("Different instance");
        }
}
```

OUTPUT

```
1. 4849051
2. 4849051
3. Same instance
4.
```

Bean Scope

Bean Scope - prototype

Spring

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

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

```
<bean    id="reportService"    class="com.infosys.demo.ReportService"
scope="prototype"/>
```

You can use prototype scope in scenarios where you need **stateful beans** and use singleton scope for **stateless beans.**

Bean Scope

Bean Scope - prototype

Spring

Consider the Report Generation application, let us now define reportService bean as a prototype bean as shown below.

```
1. <bean id="reportService" class="com.infosys.demo.ReportService" scope="prototype" >
2.                 <property name="master" ref="htmlReportGenerator" />
3.                 <property name="recordsPerPage" value="500" />
4.         </bean>
5.
6. <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
7.
```

In the above example, reportService bean is defined as prototype scope and htmlReportGenerator (since attribute scope does not exist in the bean definition, by default it is a singleton) bean is in singleton scope.

Bean Scope

Bean Scope - prototype

Spring

Let us access reportService bean in the client class more than once and compare their hashCode to see whether we are getting the same bean or different bean.

```
1.  public class Client {
2.          public static void main(String[] args) {
3.                  ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
4.                  ReportService srv1 = (ReportService)context.getBean("reportService");
5.                  ReportService srv2 = (ReportService)context.getBean("reportService");
6.                  System.out.println("hash code of srv1:" + srv1.hashCode());
7.                    System.out.println("hash code of srv2:" +srv2.hashCode());
8.                  if(srv1 == srv2){
9.                          System.out.println("Same instance");
10.                 }
11.                 else
12.                         System.out.println("Different instance");
13.         }
14. }>
```

Bean Scope

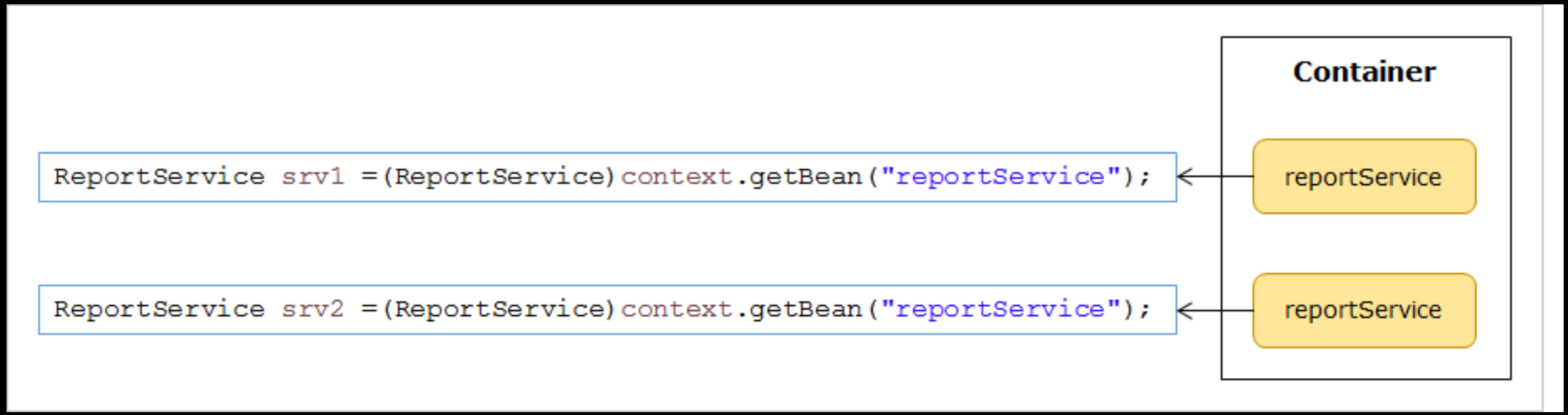
Bean Scope - prototype

Spring

Execution of the above client code gives below result.

```
1. hash code of srv1:1690287238
2. hash code of srv2:1690254271
3. Different instance
4.
```

In the above output response, you can see that hashCode of srv1 and srv2 are different indicating they are referring to different bean instances.



Bean Scope

Demo : Prototype bean scope

Spring

Highlights:

- Objective: To create a bean with prototype scope
- Required Jars: Same as Demo1

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Bean Scope

Demo : Prototype bean scope

Spring

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}

```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}

```

Bean Scope

Demo : Prototype bean scope

Spring

ReportService.java --> Service class

```
package com.infosys.demo;
public class ReportService {
    private ReportGenerator master;
    private int recordsPerPage;
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    public void setMaster(ReportGenerator master) {
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Bean Scope

Demo : Prototype bean scope

Spring

config.xml--> Spring configuration in XML

```xml
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="reportService" class="com.infosys.demo.ReportService" scope="prototype" >
        <property name="master" ref="htmlReportGenerator" />
        <property name="recordsPerPage" value="500" />
    </bean>

    <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />

    <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />



</beans>
```

Bean Scope

Demo : Prototype bean scope

Spring

Client.java --> Client Code

```
package com.infosys.demo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Client {
    private static ApplicationContext context;
    public static void main(String[] args) {
        context = new ClassPathXmlApplicationContext("config.xml");
        ReportService srv1 = (ReportService)context.getBean("reportService");
        ReportService srv2 = (ReportService)context.getBean("reportService");
        System.out.println("hash code of srv1:" + srv1.hashCode());
        System.out.println("hash code of srv2:" +srv2.hashCode());
        if(srv1==srv2){
            System.out.println("Same instance");
        }
        else
            System.out.println("Different instance");
    }
}
```

OUTPUT

```
1. hash code of srv1:1690287238
2. hash code of srv2:1690254271
3. Different instance
4.
```

Introduction to Spring Annotation Based Configuration

Spring

1. Introduction to Spring Framework
2. Spring Inversion of Control (IoC)
3. Dependency Injection (DI)
4. Autowiring
5. Bean Scope
6. Spring Annotation and Java based configuration
   - 6.1 Introduction to Annotation configuration
     - Autowiring using annotation
     - Auto scan
   - 6.2 Introduction to Java based configuration
7. Spring AOP

Introduction to Spring Annotation Based Configuration

Introduction to Autowired Annotation

Spring

Spring Annotation Based Configuration

Till now we have been using XML for writing Spring configurations. Spring also supports using **Annotations** for configuration. And the best part is that, both of them can also co-exist in the same project.

In Annotation based configuration, configuration is provided by using annotations on the relevant property, method or the class.

Annotation based configuration reduces the amount of configuration required in the XML file by using

- Autowiring: This supports autowiring dependencies using annotations such as @Autowired and @Qualifier.
- Auto Scanning: This enable Spring to auto create beans of the required classes and register with container for usage by eliminating the explicit <bean> definitions in the XML file.

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Enabling Annotation Support in Spring

By default autowiring using annotations is not turned on, <context:annotation-config> tag has to be specified in XML-based configuration to enable annotations support.

The XML configuration looks like as follows:

```
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config /> 2
</beans>
```

1

1. spring "context" namespace has "annotation-config" tag and hence context namespace has to be included in the declaration
2. This will enable support for annotations

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Introduction to @Autowired Annotation

@Autowired annotation perform byType Autowiring i.e. dependency is injected based on bean type

@Autowired cannot be used for primitive values.

This annotation can be used at property, setter method or constructor level

Autowiring the property

```
@Autowired
private ReportGenerator report;
```

- @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
 setReport(ReportGenerator report)
{
        this.report = report;
}
```

- @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
 ReportService(ReportGenerator report)
{
        this.report = report;
}
```

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

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

@Autowired On Property

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

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

```
1. package com.infosys.demo;
2. public class ReportService {
3.         @Autowired
4.         private ReportGenerator report;
5.
6.         ------------------------------------
7.
8.         }
```

@Autowired is by default wire the dependency based on type of bean. Hence it is same as autowired byType mode which is achieved using annotation @Autowired.

Introduction to Spring Annotation Based Configuration

Introduction to Autowired Annotation

Spring

Introduction to @Autowired Annotation

Following are the required configuration.

```
1.  <beans          xmlns="http://www.springframework.org/schema/beans"
2.          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.             xmlns:context="http://www.springframework.org/schema/context"
4.             xsi:schemaLocation="http://www.springframework.org/schema/beans
5.             http://www.springframework.org/schema/beans/spring-beans.xsd
6.             http://www.springframework.org/schema/context
7.             http://www.springframework.org/schema/context/spring-context.xsd">
8.          <!-- To enable annotation support in Spring-->
9.          <context:annotation-config />
10.
11.         <bean id="reportService" class="com.infosys.demo.ReportService">
12.                         <property name="recordsPerPage" value="500" />
13.             </bean>
14.
15.             <bean id="pdfReportGenerator"
    class="com.infosys.demo.PDFReportGenerator" />
16.
17. </beans>
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Introduction to @Autowired Annotation

Look at the reportService bean definition in the above configuration, there is no need of explicit attribute autowire and also no explicit bean dependencies as @Autowired used on property automatically wire the dependency based on bean type.

@Autowired annotation can be used on

- parameterized constructor for constructor injection
- setter method for setter injection

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired On Property

Highlights:

- Objective: To use Annotation for autowiring properties
- Required Jars : Jars used in the previous demo and Spring-aop-5.0.5.RELEASE.jar

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired On Property

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}
```

Introduction to Spring Annotation

Introduction to Autowired Annotat

ReportService.java --> Service class

```java
public class ReportService {
    @Autowired  // For initializing only object dependency
    private ReportGenerator master;

    @Value("100") // Annotation for initializing primitive types
    private int recordsPerPage;
    public ReportService() {
        System.out.println("default constructor");
    }
    public ReportService(ReportGenerator master) {
        System.out.println("constructor");
        this.master = master;
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    public void setMaster(ReportGenerator master) {
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired On Property

config.xml--> Spring configuration in XML

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <beans xmlns="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.      xmlns:context="http://www.springframework.org/schema/context"
6.      xsi:schemaLocation="http://www.springframework.org/schema/beans
7.         http://www.springframework.org/schema/beans/spring-beans.xsd
8.         http://www.springframework.org/schema/context
9.         http://www.springframework.org/schema/context/spring-context.xsd">
10.
11.     <context:annotation-config />
12.
13.     <bean id="reportService" class="com.infosys.demo.ReportService" />
14.
15.     <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
16.
17. </beans>
18.
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired On Property

Client.java --> Client Code

```java
package com.infosys.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
        ReportService srv = (ReportService)context.getBean("reportService");
        srv.generateReport();

    }

}
```

OUTPUT

```
default constructor
Generated PDF Report with 100 records

```

Spring

Introduction to Spring Annotation Based Configuration

Introduction to Autowired Annotation

Demo : Autowired On Constructor

Highlights:

- Objective: To use annotation for autowiring the constructor
- Required Jars: Same as previous demo

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired On Constructor

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}
```

AUK

Introduction to Spring Annot

Introduction to Autowired An

ReportService.java --> Service class

```
}
}
```

```java
package com.infosys.demo;
import org.springframework.beans.factory.annotation.Autowired;
public class ReportService {
    private ReportGenerator master;
    private int recordsPerPage;
    public ReportService() {
        System.out.println("default constructor");
    }
    @Autowired
    public ReportService(int recordsPerPage,ReportGenerator master) {
        System.out.println("Parameterized Constructor");
        this.recordsPerPage = recordsPerPage;
        this.master = master;
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        System.out.println("Setter method of RecordsPerPage property");
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    public void setMaster(ReportGenerator master) {
        System.out.println("Setter method of master property");
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
```

Introduction to Spring Annotation Based Configuration

Introduction to Autowired Annotation

Spring

Demo : Autowired On Constructor

config.xml--> Spring configuration in XML

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <beans xmlns="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
5.      xsi:schemaLocation="http://www.springframework.org/schema/beans
6.         http://www.springframework.org/schema/beans/spring-beans.xsd
7.         http://www.springframework.org/schema/context
8.         http://www.springframework.org/schema/context/spring-context.xsd">
9.
10.     <context:annotation-config />
11.
12.     <bean id="reportService" class="com.infosys.demo.ReportService">
13.         <constructor-arg name="recordsPerPage" value="150" />
14.     </bean>
15.
16.     <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
17. </beans>
18.
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired On Constructor

Client.java --> Client Code

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6.  public class Client {
7.
8.      public static void main(String[] args) {
9.          ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
10.         ReportService srv = (ReportService)context.getBean("reportService");
11.         srv.generateReport();
12.
13.     }
14.
15. }
```

OUTPUT

```
1.  Parameterized Constructor
2.  Generated PDF Report with 150 records
3.
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired on Setter Method

Highlights:

- To understand @Autowired usage on setter method
- Required Jars: Same as previous demo

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired on Setter Method

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}

```

AUK

Introduction to Spring Annot

Introduction to Autowired An

ReportService.java --> Service class

```java
public class ReportService {
    private ReportGenerator master;
    private int recordsPerPage;
    public ReportService() {
        System.out.println("default constructor");
    }
    public ReportService(int recordsPerPage,ReportGenerator master) {
        System.out.println("Parameterized Constructor");
        this.recordsPerPage = recordsPerPage;
        this.master = master;
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        System.out.println("Setter method of RecordsPerPage property");
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    @Autowired
    public void setMaster(ReportGenerator master) {
        System.out.println("Setter method of master property");
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired on Setter Method

config.xml--> Spring configuration in XML

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <beans xmlns="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.      xmlns:context="http://www.springframework.org/schema/context"
6.      xsi:schemaLocation="http://www.springframework.org/schema/beans
7.         http://www.springframework.org/schema/beans/spring-beans.xsd
8.         http://www.springframework.org/schema/context
9.         http://www.springframework.org/schema/context/spring-context.xsd">
10.
11.     <context:annotation-config />
12.
13.     <bean id="reportService" class="com.infosys.demo.ReportService">
14.         <property name="recordsPerPage" value="150" />
15.     </bean>
16.
17.     <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
18.
19. </beans>
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Demo : Autowired on Setter Method

Client.java --> Client Code

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6.  public class Client {
7.
8.      public static void main(String[] args) {
9.          ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
10.         ReportService srv = (ReportService)context.getBean("reportService");
11.         srv.generateReport();
12.
13.     }
14.
15. }
16.
```

OUTPUT

```
1.  default constructor
2.  Setter method of master property
3.  Setter method of RecordsPerPage property
4.  Generated PDF Report with 150 records
5.
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Autowired Annotation

Autowiring - Best Practice

It is generally advisable to autowire beans of workflow classes such as controller, service or repository and not advisable to autowire domain objects as they represent data in a table.

Let us look at the previously implemented Employee application.

Here if you observe,

- EmployeeRepository and LoanService dependency has been autowired in EmployeeServiceImpl class
- LoanRepository has been autowired in LoanServiceImpl class
- In the getEmployeeDetails() method of EmployeeRepositoryImpl class, required domain objects of type Employee and Address have been instantiated using the new operator as they represent data from the database.

Introduction to Spring Annotation Based Configuration

Introduction to Qualifier Annotation

Spring

Introduction to @Qualifier Annotation

As we learnt, @Autowired by default resolve bean dependencies by type.

How do we make @Autowired to work by name to allow the scenarios with more than one bean of same type in the configuration?

Consider the Report Generation application to understand this scenario.

Following is the configuration where in we have two beans **htmlReportGenerator** and **pdfReportGenerator** of same interface type ReportGenerator.

Introduction to Spring Annotation Based Configuration

Introduction to Qualifier Annotation

Spring

Introduction to @Qualifier Annotation

```
<beans          xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

        <context:annotation-config />
        <bean id="reportService" class="com.infosys.demo.ReportService" />
            <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
            <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />

</beans>
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Qualifier Annotation

Introduction to @Qualifier Annotation

In the ReportService class, we have used @Autowired on ReportGenerator property of EmployeeServiceImpl class.

```
1. package com.infosys.demo;
2. public class ReportService {
3.         @Autowired
4.         private ReportGenerator master;
5.         //@Value annotation is used to provide initialization values for primitive types
6.         @Value("100")
7.         private int recordsPerPage;
8.         ----------------------------------
9. }
```

In this scenario, application throws an exception "**UnsatisfiedDependencyException**" for ReportGenerator dependency.

Why the application is throwing **UnsatisfiedDependencyException**?

As you know, @Autowired annotation wire the dependency based on type of bean and in byType there has to be only one bean of specified type in the configuraiton but in the above specified scenario, there are two beans(**htmlReportGenerator and pdfReportGenerator)** of type ReportGenerator and hence there is an ambiguity on which bean to wire causing the application to throw an exception.

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Qualifier Annotation

Introduction to @Qualifier Annotation

How do we resolve this exception to make one of the bean to be wired for the dependency?

@Qualifier("beanId") annotation can be used along with @Autowired annotation to achieve this requirement.

In scenarios, where there are more than one bean of the same type exists in the container, @Qualifier specify which bean has to be wired based on mentioned bean id.

Let us look at the modified ReportService class with required changes to wire ReportGenerator dependency with the pdfReportGenerator bean.

```
1.  package com.infosys.demo;
2.
3.  public class ReportService {
4.          @Autowired
5.          @Qualifier("pdfReportGenerator")
6.          private ReportGenerator master;
7.
8.          @Value("100")
9.          private int recordsPerPage;
10.
11.         ----------------------------------
12. }
```

Use of @Qualifier will make @Autowired to resolve dependency based on the name of bean id specified with @Qualifier.

Introduction to Spring Annotation Based Configuration

Spring

Demo : Autowired and Qualifier

@Autowired and @Qualifier

Highlights:

- How to use @Qualifier along with @Autowired to specify the bean id to autowire
- Required Jars: Same as the previous demo

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Spring Annotation Based Configuration

Spring

Demo : Autowired and Qualifier

@Autowired and @Qualifier

Highlights:

- How to use @Qualifier along with @Autowired to specify the bean id to autowire
- Required Jars: Same as the previous demo

Demosteps:

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Spring Annotation Based Configuration

Demo : Autowired and Qualifier

Spring

@Autowired and @Qualifier

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}
```

AUK

Introduction to Spring Annotation E

Demo : Autowired and Qualifier

ReportService.java --> Service class

```java
package com.infosys.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
public class ReportService {

    @Autowired
    @Qualifier("htmlReportGenerator")
    private ReportGenerator master;

    @Value("100")
    private int recordsPerPage;
    public ReportService() {
        System.out.println("default constructor");
    }

    public ReportService(ReportGenerator master) {
        System.out.println("constructor");
        this.master = master;
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        System.out.println("setRecordsPerPage");
        this.recordsPerPage = recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
```

Introduction to Spring Annotation Based Configuration

Spring

Demo : Autowired and Qualifier

@Autowired and @Qualifier

ReportService.java --> Service class

```
    public void setMaster(ReportGenerator master) {
        System.out.println("setMaster");
        this.master = master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Introduction to Spring Annotation Based Configuration

Spring

Demo : Autowired and Qualifier

@Autowired and @Qualifier

config.xml--> Spring configuration in XML

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <beans xmlns="http://www.springframework.org/schema/beans"
4.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
5.      xsi:schemaLocation="http://www.springframework.org/schema/beans
6.         http://www.springframework.org/schema/beans/spring-beans.xsd
7.         http://www.springframework.org/schema/context
8.         http://www.springframework.org/schema/context/spring-context.xsd">
9.
10.     <context:annotation-config />
11.
12.     <bean id="reportService" class="com.infosys.demo.ReportService" />
13.
14.     <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
15.
16.     <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
17.
18. </beans>
```

Introduction to Spring Annotation Based Configuration

Spring

Demo : Autowired and Qualifier

@Autowired and @Qualifier

Client.java --> Client Code

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6.  public class Client {
7.      public static void main(String[] args) {
8.          ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
9.          ReportService srv = (ReportService)context.getBean("reportService");
10.         srv.generateReport();
11.     }
12. }
```

OUTPUT

```
1.  default constructor
2.  Generated HTML Report  with 100 records
3.
```

Spring

Introduction to Spring Annotation Based Configuration

Introduction to Spring Java Based Configuration

Annotation based configuration needs the existence of a XML file with required declarations for enabling auto scan feature of Spring.

We can eliminate the need for a XML file completely and provide the same configurations in a **Java class file.**

Configurations in a XML file and the configurations in a Java class can co-exist as well.

In Java based configuration, we need to use @Configuration and @Bean annotations in the Java class to provide bean definitions.

Spring

Introduction to Spring Annotation Based Configuration

Introduction to Spring Java Based Configuration

@Configuration

It indicates Spring container that the class contains the bean definitions.

This is a pure Java approach to configure Spring container.

@Bean

@Bean is used for defining a bean in Java based configuration.

In @Configuration annotated Java class, methods annotated with @Bean provides the bean definition.

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Spring Java Based Configuration

Spring processes @Bean annotated method and register the object returned by this method with Spring container.

By default, bean name is the same as the method name.

```
1 @Configuration
  public class AppConfig
  {
        @Bean
        public ReportService reportService()
    2   {
              return new ReportService();
        }
  }


  // ApplicationContext instantiation                                    3
  ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
  ReportService service = (ReportService) context.getBean("reportService");
```

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Spring Java Based Configuration

1. @Configuration indicates Spring that this class has bean definitions
2. ReportService class bean definition. In this example, the bean id is "reportService" which is the default name (same as the method name)
3. A class annotated with @Configuration is used as input for installing an ApplicationContext through AnnotationConfigApplicationContext class

Introduction to Spring Annotation Based Configuration

Spring

Introduction to Spring Java Based Configuration

Bean id can be specified explicitly in @Bean annotation using the "name" attribute. Explicit bean name overrides default name.

The bean scope also can be defined for a bean using @Scope annotation in Java configuration.

```
@Configuration
public class AppConfig
{
        1  @Bean("repoService")
        2  @Scope("prototype")
           public ReportService reportService()
           {
                    return new ReportService();
           }
}
```

1. Bean id is explicitly specified as "repoService", which overrides the default name
2. Bean scope is specified as "prototype"

Introduction to Spring Annotation Based Configuration

Introduction to Spring Java Based Configuration

Let us now understand how the configuration metadata can be provided through **Java class** and eliminate the need for an XML file completely.

Consider the Report Generation application Spring configuration provided through XML configuration as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- htmlReportGenerator bean definition with bean definition-->
    <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
    <!-- pdfReportGenerator bean definition with bean definition-->
    <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />
   <!-- reportService bean definition with bean dependencies through constructor injection -->
   <bean id="reportService1" class="com.infosys.demo.ReportService">
            <constructor-arg ref="pdfReportGenerator" />
            <constructor-arg value="150" />
   </bean>
  <!-- reportService bean definition with bean dependencies through setter injection -->
  <bean id="reportService2" class="com.infosys.demo.ReportService">
          <property name="master" ref="HTMLReportGenerator" />
          <property name="recordsPerPage" value="150" />
 </bean>
</beans>
```

AUK

Intro

Let us now replace above XML configuration with Spring Java based configuration as shown below through a JAVA class.

Introduction to Spring Java Based Configuration

```
package com.infosys.demo;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
        @Bean // htmlReportGenerator bean definition with bean definition
        public ReportGenerator htmlReportGenerator(){
                return new HTMLReportGenerator();
        }

        @Bean // pdfReportGenerator bean definition with bean definition
        public ReportGenerator pdfReportGenerator(){
                return new PDFReportGenerator();
        }

        @Bean // reportService bean definition with bean dependencies through constructor injection
        public ReportService reportService1(){
                ReportService reportService=new ReportService(pdfReportGenerator(), 150);
                return  reportService;
        }
    @Bean // reportService bean definition with bean dependencies through setter injection
    public ReportService reportService2(){
          ReportService reportService=new ReportService();
          reportService.setMaster(htmlReportGenerator());
          reportService.setRecordsPerPage("150");
          return reportService;
    }
}
```

Constructor Injection

Setter injection

Introduction to Spring Annotation Based Configuration

Introduction to Spring Java Based Configuration

Now because the configuration is provided through Java class, ApplicationContext in client code is instantiated using AnnotationConfigApplicationContext implementation by providing configuration class as argument.

```
1. public class EmployeeClient {
2.
3.         /** The ApplicationContext instantiation */
4.         ApplicationContext context = new
   AnnotationConfigApplicationContext(AppConfig.class);
5.
6.     ------------------------
7. }
```

Spring

Introduction to Spring Annotation Based Configuration

Demo : Spring Java Based Configuration

Highlights:

- To understand Java code based configuration in Spring
- To understand the application container instantiation for Java configuration

Demosteps:

Required Jars

Following are the Spring jars(same as previous demo) required for this example

- spring-beans.jar
- spring-core.jar
- spring-context.jar
- spring-expression.jar
- spring-aop.jar
- spring-jcl.jar

Introduction to Spring Annotation Based Configuration

Spring

Demo : Spring Java Based Configuration

Earlier to Spring 3.2, it is necessary to add following jars explicitly to your application classpath to process @Configuration annotated classes:

- Spring-asm.jar
- cglib-nodep.jar

But from Spring 3.2 onwards, **spring-core.jar** itself include these functionalities.

ReportGenerator.java--> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Spring Annotation Based Configuration

Spring

Demo : Spring Java Based Configuration

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

@Component(value="htmlGenerator")
public class HTMLReportGenerator implements ReportGenerator{
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated HTML Report with " + recordsPerPage + " records";
	}
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

@Component(value="pdfGenerator")
public class PDFReportGenerator implements ReportGenerator {
	@Override
	public String generateReport(int recordsPerPage) {
		return "Generated PDF Report with " + recordsPerPage + " records";
	}

}
```

Introduction to Spring Anno

Demo : Spring Java Based Co

ReportService.java --> Service class

```java
package com.infosys.demo;
public class ReportService {
    private ReportGenerator master;
    private int recordsPerPage;
    public void setMaster(ReportGenerator master) {
            this.master = master;
    }
    public void setRecordsPerPage(int recordsPerPage) {
            this.recordsPerPage = recordsPerPage;
    }
    public ReportService() {
        System.out.println("constructor");
    }
    public ReportService(ReportGenerator master, int recordsPerPage) {
        super();
        System.out.println("Parameterized Constructor");
        this.master = master;
        this.recordsPerPage = recordsPerPage;
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
    public ReportGenerator getMaster() {
        return master;
    }
    public void generateReport() {
        System.out.println(master.generateReport(recordsPerPage));
    }
}
```

Introdu

Demo

AppConfig.java--> Spring configuration

```java
package com.infosys.demo;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean // htmlReportGenerator bean definition with bean definition
    public ReportGenerator htmlReportGenerator() {
        return new HTMLReportGenerator();
    }
    @Bean // pdfReportGenerator bean definition with bean definition
    public ReportGenerator pdfReportGenerator() {
        return new PDFReportGenerator();
    }
    @Bean // reportService bean definition with bean dependencies through constructor injection
    public ReportService reportService1() {
        ReportService reportService = new ReportService(pdfReportGenerator(), 150);
        return reportService;
    }
    @Bean // reportService bean definition with bean dependencies through setter injection
    public ReportService reportService2() {
        ReportService reportService = new ReportService();
        reportService.setMaster(htmlReportGenerator());
        reportService.setRecordsPerPage(150);
        return reportService;
    }
}
```

Introduction to Spring Annotation Based Configuration

Demo : Spring Java Based Configuration

Client.java --> Client Code

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5.
6.  public class Client {
7.
8.      public static void main(String[] args) {
9.          ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
10.         ReportService srv1 = (ReportService)context.getBean("reportService1");
11.         srv1.generateReport();
12.         ReportService srv2 = (ReportService)context.getBean("reportService2");
13.         srv2.generateReport();
14.     }
15. }
16.
```

OUTPUT

```
1.  Parameterized Constructor
2.  constructor
3.  Generated PDF Report with 150 records
4.  Generated HTML Report  with 150 records
```

Introduction to Spring Annotation Based Configuration

Spring

Java Code Based Configuration - Advantages

Benefits of Java based configuration:

- XML parser is not required as the entire configuration is now provided through Java code
- It does compile time check which helps in debugging the code easily
- This support easy refactoring of the code
- It simplify test automation

Introduction to Spring Annotation Based Configuration

Spring

@ComponentScan and @PropertySource in Spring Java Based Configuration

Let us look at the **compile time check** in detail.

Consider the below code scenario to understand it better.

```
1.  public class EmployeeServiceImpl {
2.
3.      //dependent on EmployeeRepository bean
4.      private EmployeeRepository employeeRepository;
5.
6.      public EmployeeServiceImpl(EmployeeRepository employeeRepository) {
7.          this.employeeRepository = employeeRepository;
8.      }
9.
10.     -------
11. }
12.
```

Introduction to Spring Annotation Based Configuration

Spring

@ComponentScan and @PropertySource in Spring Java Based Configuration

Below given is the Spring configuration to define the required beans.

```
1.  <!-- service bean definition -->
2.  <bean id="employeeService" class="com.infosys.service.EmployeeServiceImpl">
3.          <!-- Trying to inject loanRepositoy bean to employeeRepository property
    which is not a valid type of injection -->
4.          <property name="employeeRepository" ref="loanRepository" />
5.  </bean>
6.
7.  <!-- repository bean definition -->
8.  <bean id="loanRepository" class="com.infosys.repository.LoanRepositoryImpl">
    </bean>
9.  <bean id="employeeRepository"
    class="com.infosys.repository.EmployeeRepositoryImpl"></bean>
10.
```

Above code does not show any compilation error but at **runtime** it throws **BeanCreationException** (because employeeService need an employeeRepository bean and not a loanRepository bean).

Introduction to Spring Annotation Based Configuration

Spring

@ComponentScan and @PropertySource in Spring Java Based Configuration

Equivalent **Java configuration** does not show this problem. It shows compilation error at that specific line in the code, you cannot compile without resolving the issue.

```
@Configuration
public class AppConfig {
    @Bean
    EmployeeServiceImpl employeeService(){
        EmployeeServiceImpl employeeService = new EmployeeServiceImpl();
        // Shows compilation error at this line
        employeeService.setEmployeeRepository(loanRepository());
        return employeeService;
    }
    @Bean
    LoanRepositoryImpl loanRepository(){
        LoanRepositoryImpl loanRepository = new LoanRepositoryImpl();
        return loanRepository;
    }
    @Bean
    EmployeeRepositoryImpl employeeRepository(){
        EmployeeRepositoryImpl employeeRepository = new EmployeeRepositoryImpl();
        return employeeRepository;
    }
}
```

Spring

Introduction to Spring Annotation Based Configuration

@ComponentScan and @PropertySource in Spring Java Based Configuration

Java Configuration - Auto Scan Feature

As part of the annotation-based configuration, we learned how to eliminate explicit bean definitions by using auto scan feature of Spring.

Does Java-based configuration support auto scan feature of Spring?

Yes, it supports auto scan feature of Spring using the annotation **@ComponentScan.**

Consider the Report Generation application configuration provided through Java-based configuration as shown below.

Introduction to Spring Annotation Based Configuration

Spring

@ComponentScan and @PropertySource in Spring Java Based Configuration

Java Configuration - Auto Scan Feature

```
package com.infosys.demo;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {

    @Bean // htmlReportGenerator bean definition with bean definition
    public ReportGenerator htmlReportGenerator(){
        return new HTMLReportGenerator();
    }

    @Bean // pdfReportGenerator bean definition with bean definition
    public ReportGenerator pdfReportGenerator(){
        return new PDFReportGenerator();
    }
    // reportService bean definition with bean dependencies through constructor injection
    @Bean
    public ReportService reportService(){
        ReportService reportService=new ReportService(pdfReportGenerator());
        return  reportService;
    }
}
```

Introduction to Spring Annotation Based Configuration

Spring

@ComponentScan and @PropertySource in Spring Java Based Configuration

Java Configuration - Auto Scan Feature

Let us now introduce auto scan feature in Java configuration allowing Spring to auto create beans instead of explicit bean definition as shown below.

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.annotation.Bean;
4.  import org.springframework.context.annotation.Configuration;
5.
6.  @Configuration
7.  @ComponentScan("com.infosys.demo")
8.  public class AppConfig
9.  {
10.
11. }
```

Spring

Introduction to Spring Annotation Based Configuration

@ComponentScan and @PropertySource in Spring Java Based Configuration

@PropertySource

You know that the Properties file can be read using Properties class.

However, Spring provides Environment class to directly access the values in the properties file. In order to access the properties, the Environment class' bean can be autowired as follows:

```
1. @Autowired
2. Environment env;
3.
```

Using the env bean, the property values can be fetched using the getProperty() method as follows:

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

Spring

Introduction to Spring Annotation Based Configuration

@ComponentScan and @PropertySource in Spring Java Based Configuration

@PropertySource

The location of the properties file is mentioned in the Spring configuration file using @PropertySource annotation along with @Configuration annotation. For example, application.properties which is present in classpath can be loaded using @PropertySource as follows:

```
1. @Configuration
2. @PropertySource("classpath:application.properties")
3.  public class SpringConfig {
4.
5.      //code for configuring other beans
6.  }
7.
```

Introduction to Spring Annotation Based Configuration

Spring

Demo: @ComponentScan and @Propertysource in Spring Java Based Configuration

@PropertySource annotation

Highlights:

To understand how to access the values from the properties file using @PropertySource annotation

Demosteps:

Required Jars: Same as the previous demo.

ReportGenerator.java --> Interface

```
1. package com.infosys.demo;
2.
3. public interface ReportGenerator {
4.         public String generateReport(int recordsPerPage);
5. }
6.
```

Introduction to Spring Annotation Based Configuration

Spring

Demo: @ComponentScan and @Propertysource in Spring Java Based Configuration

HTMLReportGenerator.java --> Implementation of ReportGenerator interface

@PropertySource annotation

```java
package com.infosys.demo;

@Component(value="htmlGenerator")
public class HTMLReportGenerator implements ReportGenerator{
        @Override
        public String generateReport(int recordsPerPage) {
                return "Generated HTML Report with " + recordsPerPage + " records";
        }
}
```

PDFReportGenerator.java --> Implementation of ReportGenerator interface

```java
package com.infosys.demo;

@Component(value="pdfGenerator")
public class PDFReportGenerator implements ReportGenerator {
        @Override
        public String generateReport(int recordsPerPage) {
                return "Generated PDF Report with " + recordsPerPage + " records";
        }

}
```

Introduction to Spring Annot

Demo: @ComponentScan and

ReportService.java --> Service class

@PropertySource annotation

```java
package com.infosys.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
@Service
public class ReportService {
    @Autowired
    @Qualifier("pdfGenerator")
    private ReportGenerator master;
    @Value("0")
    private int recordsPerPage;
    public void setMaster(ReportGenerator master) {
        this.master = master;
    }
    public void setRecordsPerPage(int recordsPerPage) {
        this.recordsPerPage = recordsPerPage;
    }
    public ReportService() {
    }
    public ReportService(ReportGenerator master, int recordsPerPage) {
        super();
        this.master = master;
        this.recordsPerPage = recordsPerPage;
    }
    public int getRecordsPerPage() {
        return recordsPerPage;
    }
```

Spring

Introduction to Spring Annotation Based Configuration

Demo: @ComponentScan and @Propertysource in Spring Java Based Configuration

ReportService.java --> Service class

@PropertySource annotation

```
    public ReportGenerator getMaster() {
        return master;
    }
    public void generateReport() throws ReportGeneratorException {
        try {
            System.out.println(master.generateReport(recordsPerPage));
        } catch (ReportGeneratorException reportGeneratorException) {
            System.out.println(reportGeneratorException.getMessage());
        }
    }
}
```

Introduction to Spring Annotation Based Configuration

Spring

Demo: @ComponentScan and @Propertysource in Spring Java Based Configuration

@PropertySource annotation

AppConfig.java --> Spring configuration

```
1.  package com.infosys.demo;
2.
3.  import org.springframework.context.annotation.ComponentScan;
4.  import org.springframework.context.annotation.Configuration;
5.  import org.springframework.context.annotation.PropertySource;
6.
7.  @Configuration
8.  // The annotation used below is to specify the location of the properties file
9.  @PropertySource("classpath:application.properties")
10. @ComponentScan("com.infosys.demo")
11. public class AppConfig {
12.
13. }
14.
```

Introduction to Spring Annotation Based Configuration

Spring

Demo: @ComponentScan and @Propertysource in Spring Java Based Configuration

Client.java --> Client Code

@PropertySource annotation

```java
package com.infosys.demo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Client {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        ReportService srv1 = (ReportService)context.getBean("reportService1");
        srv1.generateReport();

    }
}
```

application.properties

```
1. Insufficient_records="Insufficient number of records"
```

OUTPUT

```
1. "Insufficient number of records"
```

Spring

SpringBoot Logging

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.

Spring

SpringBoot Logging

Logging - Levels of Logging

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

Spring

SpringBoot Logging

Logging

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'
```

Spring

SpringBoot Logging

Logging

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.

Spring

SpringBoot Logging

Logging - Log the error messages

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.

Spring

SpringBoot Logging

Logging - Log the error messages

```
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. }
```

Spring

SpringBoot Logging

Logging - Log the error messages

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

Spring

SpringBoot Logging

Logging - Log the error messages using Logback

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
```

Spring

SpringBoot Logging

Logging - Log the error messages using Logback

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
```

Spring

SpringBoot Logging

Logging - Log the error messages using Logback

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

Spring

SpringBoot Logging

Logging - Log

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

Spring

SpringBoot Logging

Demo : Logging

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. }
```

SpringBoot Logging

Demo : Logging

CustomerServiceImpl.java

```java
package com.infy.service;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infy.dto.CustomerDTO;
import com.infy.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
    private static Logger logger = LoggerFactory.getLogger(CustomerServiceImpl.class);
    @Autowired
    private CustomerRepository customerRepository;
    @Override
    public String createCustomer(CustomerDTO dto) {
        return customerRepository.createCustomer(dto);
    }
    @Override
    public String fetchCustomer() {
        return customerRepository.fetchCustomer();
    }
    @Override
    public void deleteCustomer(long phoneNumber) {
        try {
            customerRepository.deleteCustomer(phoneNumber);
        } catch (Exception e) {
            logger.info("In log Exception ");
            logger.error(e.getMessage(),e);
        }
    }
}
```

SpringBoot Logging

Demo : Logging

CustomerRepository.java

```java
package com.infy.repository;
import java.util.ArrayList;
import java.util.List;
import javax.annotation.PostConstruct;
import org.springframework.stereotype.Repository;
import com.infy.dto.CustomerDTO;
@Repository
public class CustomerRepository {
    @PostConstruct
    public void initializer()
    {
        CustomerDTO customerDTO = new CustomerDTO();
        customerDTO.setAddress("Chennai");
        customerDTO.setName("Jack");
        customerDTO.setEmail("Jack@infy.com");
        customerDTO.setPhoneNo(9951212222l);
        customers = new ArrayList<>();
        customers.add(customerDTO);
    }
    List <CustomerDTO> customers=null;
    public String createCustomer(CustomerDTO dto) {
        customers = new ArrayList<>();
        customers.add(dto);
        return "Customer added successfully"+customers.indexOf(dto);
    }
    public String fetchCustomer()  {
        return " The customer fetched  "+customers;
    }
```

Spring

SpringBoot Logging

Demo : Logging

CustomerRepository.java

```
    public void deleteCustomer(long phoneNumber) throws Exception
    {

        for(CustomerDTO customer : customers)
        {
            if(customer.getPhoneNo() == phoneNumber)
            {
                customers.remove(customer);
                System.out.println(customer.getName()+"of phoneNumber"+
                    customer.getPhoneNo()+"\t got deleted successfully");
                break;
            }
            else
                throw new Exception("Customer does not exist");
        }

    }
}
```

SpringBoot Logging

Demo : Logging

CustomerDTO.java

```java
package com.infy.dto;
public class CustomerDTO {
    long phoneNo;
    String name;
    String email;
    String address;
    public long getPhoneNo() {
        return phoneNo;
    }
    public void setPhoneNo(long phoneNo) {
        this.phoneNo = phoneNo;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getAddress() {
        return address;
    }
```

Spring

Spring

SpringBoot Logging

Demo : Logging

CustomerDTO.java

```
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public CustomerDTO(long phoneNo, String name, String email, String address) {
        this.phoneNo = phoneNo;
        this.name = name;
        this.email = email;
        this.address = address;
    }
    public CustomerDTO() {
    }
}
```

Spring

SpringBoot Logging

Demo : Logging

Demo8Application.java

```java
package com.infy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.AbstractApplicationContext;
import com.infy.service.CustomerServiceImpl;

@SpringBootApplication
public class Demo8Application {

    public static void main(String[] args) {

        CustomerServiceImpl service = null;
        AbstractApplicationContext context = (AbstractApplicationContext) SpringApplication.run(Demo8Application.class,
                args);

        service = (CustomerServiceImpl) context.getBean("customerService");

        service.deleteCustomer(11512122221);
        // service.deleteCustomer(9951212222l);
    }

}
```

Spring

SpringBoot Logging

Demo : Logging

Output:

```
 .   ____          _            __ _ _
/\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.13.RELEASE)

2020-04-07 14:50:12.615  INFO 99756 --- [           main] com.infy.Demo8Application                : Starting Demo8Application
2020-04-07 14:50:12.621  INFO 99756 --- [           main] com.infy.Demo8Application                : No active profile set,
2020-04-07 14:50:14.183  INFO 99756 --- [           main] com.infy.Demo8Application                : Started Demo8Application in
2020-04-07 14:50:14.187  INFO 99756 --- [           main] com.infy.service.CustomerServiceImpl     : In log Exception
2020-04-07 14:50:14.192 ERROR 99756 --- [           main] com.infy.service.CustomerServiceImpl     : Customer does not exist

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

Introduction to Spring AOP

Aspect Oriented Programming (AOP)

Spring

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.

AUK

Introduction to Spring AOP

Aspect Oriented Programming (AOP)

- 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.

Introduction to Spring AOP

AOP

```
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.

Introduction to Spring AOP

Aspect Oriented Programming (AOP)

Spring

- 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 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.

Spring

Introduction to Spring AOP

Aspect Oriented Programming (AOP)

- 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.

Introduction to Spring AOP

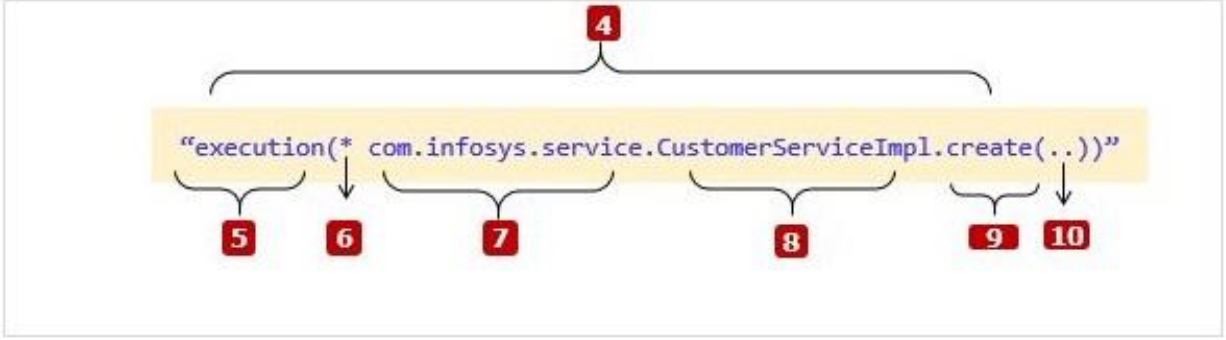
Spring

Spring AOP - Pointcut declaration

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

```
@Aspect
public class LoggingAspect
{
1  @Before( "execution(* com.infosys.service.CustomerServiceImpl.create(..))")
3  public void logging()
2  {
       // Logging code
   }
}
```

1. Advice Type
2. Advice
3. Aspect
4. Pointcut expression
5. Designator
6. Return Type : "*" refers to any return type
7. Package
8. Class
9. Method
10. Parameters: "(..)" refers to 0 or more arguments



Spring

Introduction to Spring AOP

Spring AOP - Pointcut declaration

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

Pointcut expressions have the following syntax:

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

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.

Spring

Introduction to Spring AOP

Spring AOP - Pointcut declaration

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

Spring

Introduction to Spring AOP

Configuring AOP in Spring Boot

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>
5.
```

Spring

Implementing AOP advices

Before Advice:

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:

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

Implementing AOP advices

Spring

After Advice

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.     }
```

Implementing AOP advices

After Returning Advice

Spring

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.

Implementing AOP advices

AfterThrowing Advice

Spring

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.     }
```

Implementing AOP advices

Around Advice

Spring

Around advice:

This advice gets executed around the joinpoint i.e. before and after the execution of the target method. It is declared using @Around annotation. The following is an example of this advice:

```
1. @Around("execution(* com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
2.     public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
3.         System.out.println("Before proceeding part of the Around advice.");
4.          Object cust =  joinPoint.proceed();
5.         System.out.println("After proceeding part of the Around advice.");
6.         return cust;
7.     }
```

In the above code snippet, aroundAdvice method accepts an instance of ProceedingJoinPoint as a parameter. It extends the JoinPoint interface, and it can only be used in the Around advice. The proceed() method invokes the joinpoint.

Implementing AOP advices

Demo: AOP

Highlights:

Objective: To implement AOP

Demo Steps:

LoggingAspect.java

Spring

```
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;
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.
```

Implementing AOP advices

Demo: AOP

Spring

```
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.     }
```

Implementing AOP advices

Demo: AOP

Spring

```
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. }
```

Implementing AOP advices

Demo: AOP

Spring

CustomerService.java

```java
package com.infy.service;

import java.util.List;

import com.infy.dto.CustomerDTO;

public interface CustomerService {
    public String createCustomer(CustomerDTO customerDTO);

    public List<CustomerDTO> fetchCustomer();

    public String updateCustomer(long phoneNumber, CustomerDTO customerDTO);

    public String deleteCustomer(long phoneNumber);

}

```

Spring

Implementing AOP advices

Demo: AOP

CustomerServiceImpl.java

```java
package com.infy.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.infy.dto.CustomerDTO;
import com.infy.repository.CustomerRepository;

@Service("customerService")
public class CustomerServiceImpl implements CustomerService {

    @Autowired
    private CustomerRepository customerRepository;

    public String createCustomer(CustomerDTO customerDTO) {
        customerRepository.createCustomer(customerDTO);

        return "Customer with " + customerDTO.getPhoneNo() + " added successfully";
    }

    public List<CustomerDTO> fetchCustomer() {
        // uncomment the below line to see the AfterThrowing advice
        // int b=10/0;
        return customerRepository.fetchCustomer();
    }

    public String updateCustomer(long phoneNumber, CustomerDTO customerDTO) {

        return customerRepository.updateCustomer(phoneNumber, customerDTO);
    }

    public String deleteCustomer(long phoneNumber) {

        return customerRepository.deleteCustomer(phoneNumber);
    }
}
```

Implementing AOP advices

Demo: AOP

CustomerRepository.java

```java
package com.infy.repository;

import java.util.ArrayList;
import java.util.List;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Repository;
import com.infy.dto.CustomerDTO;

@Repository
public class CustomerRepository {
    List<CustomerDTO> customers = null;

    @PostConstruct
    public void initializer() {
        CustomerDTO customerDTO = new CustomerDTO();
        customerDTO.setAddress("Chennai");
        customerDTO.setName("Jack");
        customerDTO.setEmail("Jack@infy.com");
        customerDTO.setPhoneNo(9951212222l);
        customers = new ArrayList<>();
        customers.add(customerDTO);

    }

    // adds the received customer object to customers list
    public void createCustomer(CustomerDTO customerDTO) {
        customers.add(customerDTO);
    }
```

Implementing AOP advices

Spring

Demo: AOP

```
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.
```

Implementing AOP advices

Demo: AOP

Spring

Demo9Application.java

```java
package com.infy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.AbstractApplicationContext;
import com.infy.service.CustomerServiceImpl;

@SpringBootApplication
public class Demo9Application {

    public static void main(String[] args) {

        CustomerServiceImpl service = null;
        AbstractApplicationContext context = (AbstractApplicationContext) SpringApplication.run(Demo9Application.class,
                args);
        service = (CustomerServiceImpl) context.getBean("customerService");
        service.fetchCustomer();
        context.close();
    }

}

```

Spring

Best Practices : Spring Boot Application

Best Practices

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.

Spring

Best Practices : Spring Boot Application

Use Spring Initializr for starting new Spring Boot projects

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

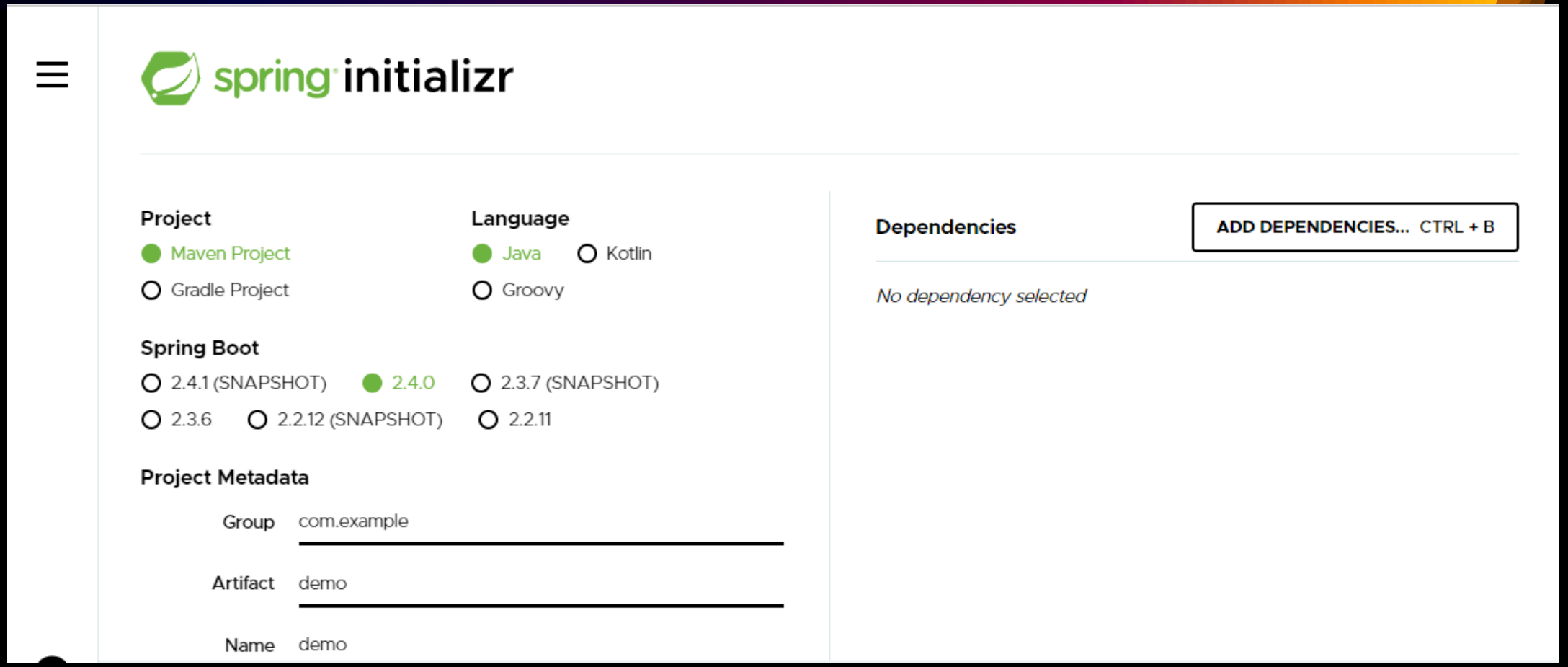
- Using Spring Initializr
- Using the Spring Tool Suite (STS)
- Using Spring Boot CLI

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.

Spring

Best Practices : Spring Boot Application

Use Spring Initializr for starting new Spring Boot projects



Spring

Best Practices : Spring Boot Application

Standard Project Structure for Spring Boot Projects

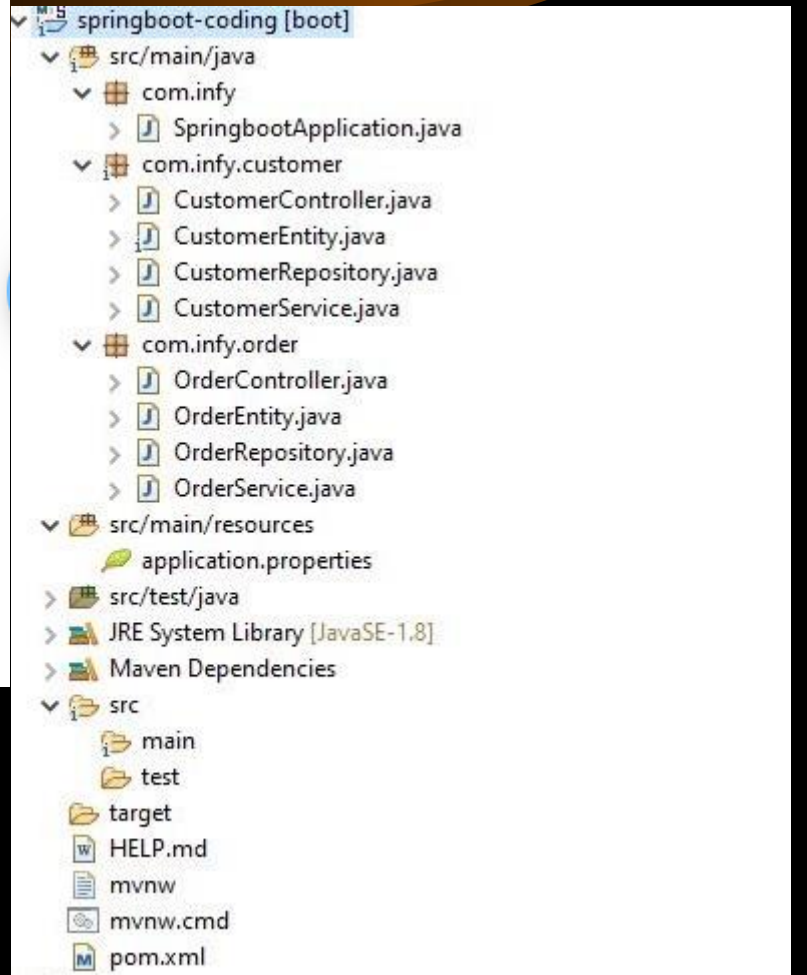
There are two recommended ways to create a spring boot project structure for Spring boot applications, which will help developers to maintain a standard coding structure.

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



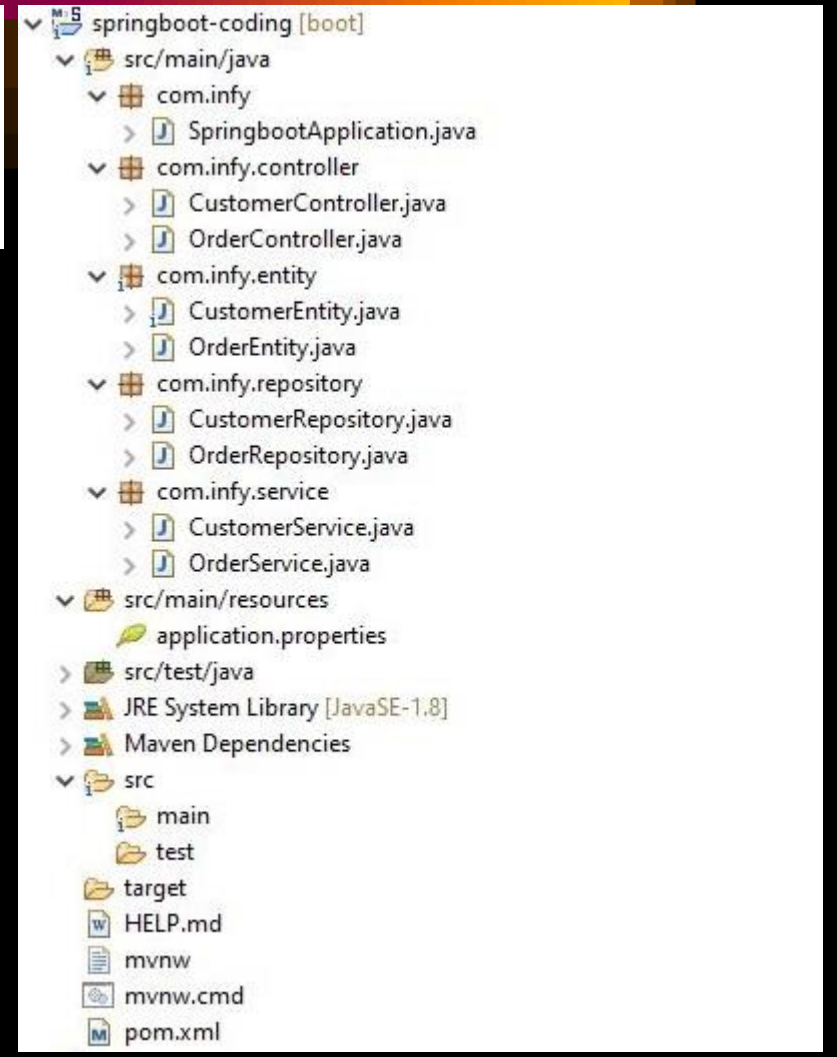
Spring

Best Practices : Spring Boot Application

Standard Project Structure for Spring Boot Projects

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.



Spring

Best Practices : Spring Application

Best Practices

- 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. }
```

Best Practices : Spring Application

Spring

Best Practices

- 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.
```

Spring

Best Practices : Spring Application

Best Practices

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

```
1. @Around("com.infy.service.CustomerServiceImpl.fetchCustomer.aspect.CommonPointConfig.logBefore
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.

Thank you