Spring Framework

Day-1

What is Spring Framework?

- **Spring is a powerful, lightweight Java framework used for building enterprise-level applications. It provides infrastructure support for developing Java applications and helps you write clean, loosely coupled, and testable code.**

Analogy: Spring Explained Simply

Imagine you're opening a donut shop:

- You need to buy flour, sugar, and oil (these are like Java classes).
- You also need to mix, fry, decorate, and serve the donuts (this is like writing code to manage those classes).

Now imagine you hire a manager (Spring):

- He knows what ingredients to use, where to get them, how to mix and cook them, and even automates the entire process.
- You just tell him, “I want chocolate donuts,” and he manages the rest.
- Spring works like this Manager. You just declare what you want, and Spring handles:
    1. Object creation (Bean)
    2. Wiring dependencies (DI)
    3. Managing configurations
    4. Integrating with databases or APIs
    5. Handling web requests and security..etc

In Simple Words:

Spring is like your assistant in Java projects. Instead of you writing everything from scratch, Spring takes care of the background work — like connecting things, managing lifecycles, and keeping your code clean and organized.

Why Should We Learn Spring Framework?

Industry Standard in Java Development

- **Spring is the most widely used Java framework for enterprise applications, web services, and microservices. Companies expect Java developers to know Spring.**

Simplifies Complex Java Code

- **Without Spring, managing dependencies, connections, configurations, security, etc., becomes repetitive and error-prone. Spring handles all that with:**

Dependency Injection (Emp-Acc)

- **Annotations**
- **Auto Configuration (in Spring Boot)**
- **Spring removes boilerplate code and lets you focus on your business logic.**

Web and REST API Development Made Easy

Spring MVC (and Spring Boot) makes it super easy to create:

- **Web Applications**
- **REST APIs**
- **JSON/XML data handling**
- **File upload/download systems**
- **Instead of managing servlets and JSPs manually, Spring abstracts everything for you.**

Built-in Security with Spring Security

Spring provides a ready-to-use security framework:

- **Role-based access**
- **JWT support**
- **Authentication & Authorization**
- **OAuth2 support**

- So you don’t need to write custom login systems from scratch.

Analogy:

- Think of Spring as a cake mix:
- Without Spring = You buy flour, sugar, eggs, bake, measure everything manually.
- With Spring = You get a pre-mix, just add water and bake!
- Spring gives you ready-made solutions for common needs (web, DB, security), so you don’t reinvent the wheel every time.

In Short:

- You should learn Spring because it’s the foundation of modern Java development — clean code, faster development, secure apps, better job opportunities.

What is Spring Framework Modules?

The Spring Framework is not a single library, but a collection of multiple modules — each designed to handle a specific task in Java application development.

Why is it modular?

- **You only include what you need.**
- **Keeps your application lightweight.**
- **Improves flexibility and performance.**

Layer	Module
Core Container	spring-core, spring-beans, spring-context, spring-expression
Data Access	spring-jdbc
Web	spring-web, spring-webmvc

NOTE:

Spring Framework is a powerful Java framework that provides several core modules like Spring Core, Spring JDBC, Spring MVC, and others. These are the official

modules that form the foundation of the Spring ecosystem. However, tools like Spring Boot, Spring Security, and concepts like Spring REST API and Microservices are not part of the core Spring Framework modules, but they are built on top of it.

Day-2

Spring Core

Birth of Spring – Early 2000s

- Inventor: Rod Johnson
- Original Work: "Expert One-on-One J2EE Design and Development" (Book, 2002)
- In his book, Rod criticized complexities in Java EE (like EJB).
- He proposed a simpler alternative — a lightweight container for managing objects and their dependencies.

- This idea became the Spring Framework.
- Spring Framework 1.0 – Released in 2004
- Initial version released by Interface21, Rod Johnson's company (later became SpringSource)

What is Spring Core?

- Spring Core provides the core functionalities of the Spring Framework, especially Dependency Injection (DI) and Inversion of Control (IOC).
- The Spring Core module is the heart of the Spring Framework. Its main job is to manage objects (called beans) in a smart, flexible, and efficient way.
- In Spring Core, the first and most important concept we learn is: How to create and manage objects using the Spring Framework instead of using the new keyword manually.

❌ Problems When Using new Keyword (in Points):

Tight Coupling

- **The class is directly dependent on a specific implementation.**
- **Difficult to change or extend later.**

No Dependency Injection

- **You must manually create and pass dependencies.**
- **Cannot leverage Spring's automatic wiring (@Autowired).**

```
public class StudentController {
    private StudentService studentService =
new StudentService();
    public void registerStudent() {
        studentService.register();
    }
}
```

Here's the problem:

StudentController is tightly coupled to StudentService.

If tomorrow you want to replace StudentService with AdvancedStudentService, you must modify the controller code:

```
private StudentService studentService =
new AdvancedStudentService(); // manual
change
```

This makes it harder to maintain, test, or scale your app.

```
public class StudentController {
    @Autowired
    private StudentService studentService;
    public void registerStudent() {
        studentService.register();
```

```
    }
}
```

Now, Spring will inject the required implementation.

If you want to use AdvancedStudentService, you just update the bean, no code change in the controller.

✅ When to Use new in a Spring Application

You can use new safely in Spring in the following situations:

1.Creating Simple Utility or Data Objects

- **Objects like new String(), new ArrayList<>(), new Date() are safe to create manually.**

2.Temporary or One-Time Use Objects

- When you need a one-time-use object inside a method, like:
- Address address = new Address("Delhi", "India");
- Scanner scan=new Scanner(System.in);

3.External Library Classes

- Classes from third-party libraries (like Apache POI, JSON parsers, etc.) often need to be created manually:
- ObjectMapper mapper = new ObjectMapper();

❗ Avoid using new for:

- Services, Repositories, Controllers, or any class that:
- Has dependencies
- Needs to be injected elsewhere

Summary:

- Use new only for lightweight, independent, temporary objects.
- For core business components, always let Spring create and manage them for you.

How to create object in Spring?

Using the Spring IoC (Inversion of Control) Container

Explanation:

- **In Spring, we don’t create objects manually using new.**
- **Instead, we register classes as beans, and the IoC Container is responsible for:**
  1. **Creating objects**
  2. **Managing their lifecycle**
  3. **Injecting their dependencies**

What is a Bean in Spring?

- **A bean is simply an object that is managed by the Spring IoC container.**
- **Think of it like this: "Bean" is just a fancy Spring name for a Java object created, managed, and injected by the Spring container.**

What is IoC Container in Spring?

- **The IoC (Inversion of Control) Container is a core part of the Spring Framework that is responsible for:**
- **Creating objects (beans), injecting dependencies, and managing their lifecycle automatically.**
- **You give control to Spring (inversion of control), and it takes care of object creation and wiring — instead of you doing it manually using new.**

IoC (Inversion of Control) - The Design Pattern Behind the Container

What is IoC?

- **IoC (Inversion of Control) is a design principle where:**

- You give control to the framework (Spring) to manage object creation and dependency injection.
- Instead of your class calling other classes (like using new), the container injects dependencies into your class automatically.

History:

- Origin: 1990s
- Philosophy: Hollywood Principle
- “Don’t call us, we’ll call you.”
- Means: You don’t create objects — the framework will do it and inject them when needed.

Types of IoC Container in Spring

- Spring provides two main IoC container interfaces for managing beans (objects):

1.BeanFactory (Older)

- **Type: Interface**
- **Introduced in: Spring 1.x**
- **Spring 3.1+ officially marked XmlBeanFactory as deprecated.**

Features:

- **Basic functionality to load and manage beans**
- **Lazy loading (beans created only when requested)**
- **Common Implementation: XmlBeanFactory (class)**

Why it’s not recommended now:

- **Limited features**
- **No support for:**
    1. **Bean auto-wiring**
    2. **Annotations**

Mostly replaced by ApplicationContext in modern Spring apps

2. ApplicationContext (Latest & Most Used)

- **Type: Interface**
- **Introduced in: Spring 2.x**

Features:

- **Extends BeanFactory (has all its features)**
- **Eager loading by default (creates all beans at startup)**
- **Common Implementation: ClassPathXmlApplicationContext (class)**

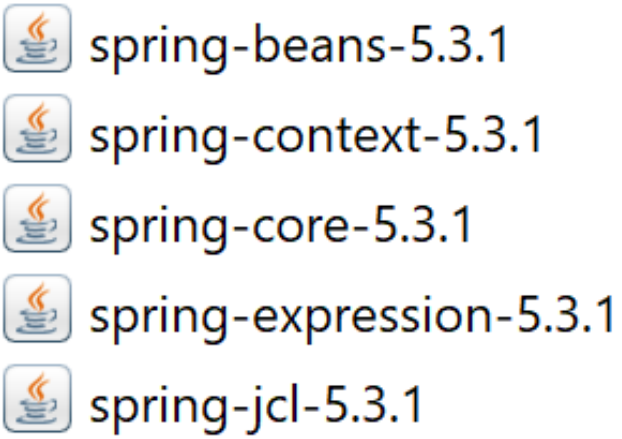
Supports:

- **Dependency Injection**
- **Annotation-based configuration**
- **Used in almost every real-world Spring project**

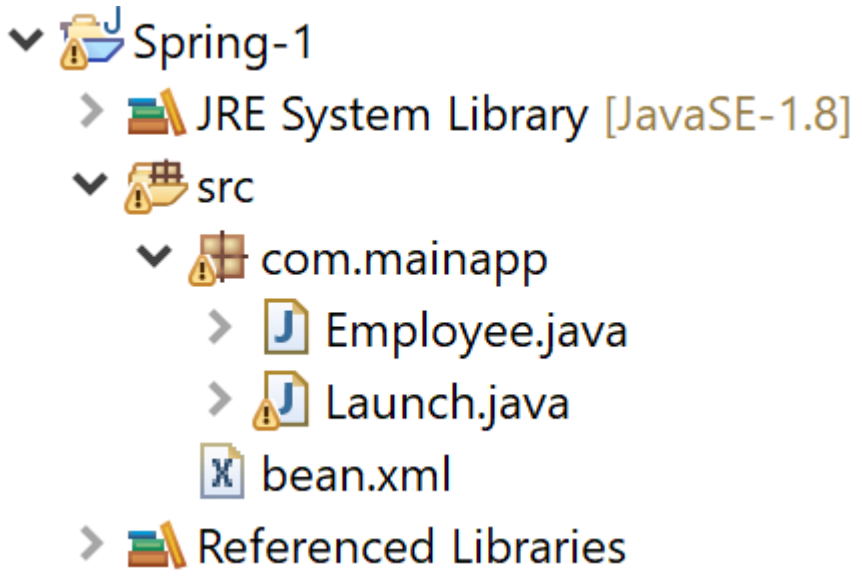
Spring util Schema

https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/xsd-configuration.html

Jar Files



==========Program: XmlBeanFactory==========



```java
package com.mainapp;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class Launch {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		ClassPathResource resource = new
ClassPathResource("bean.xml");
		XmlBeanFactory xmlBeanFactory = new
XmlBeanFactory(resource);
		Employee emp =(Employee) xmlBeanFactory.getBean("emp");
		emp.test();

	}
}

package com.mainapp;

public class Employee {

	static {
		System.out.println("EMP BEAN LOADING");
	}

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATED");
	}

	public void test() {
		System.out.println("TESTED....");
	}

}

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->

    <bean id="emp" class="com.mainapp.Employee" > </bean>
```

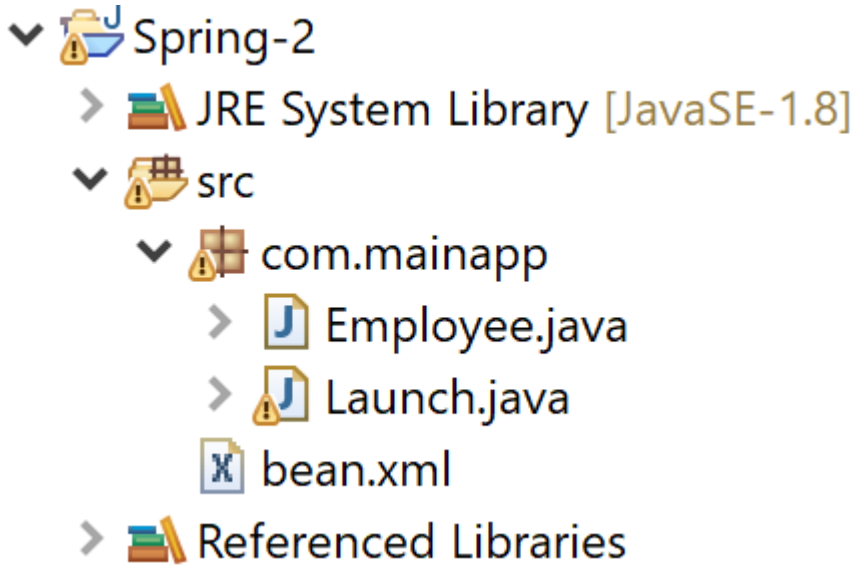
```
</beans>
```

==========

==========

==========

==========Program: ClassPathXmlApplicationCOntext==========




```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
```

```
            Employee emp =(Employee) ac.getBean("emp");
            emp.test();

      }
}
package com.mainapp;

public class Employee {

      static {
            System.out.println("EMP BEAN LOADING");
      }

      public Employee() {
            System.out.println("EMP BEAN INSTANTIATED");
      }

      public void test() {
            System.out.println("TESTED....");
      }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->

    <bean id="emp" class="com.mainapp.Employee" > </bean>

</beans>
```

==========

==========

==========

Day-3

Bean Scopes in Spring (Focus on Singleton and Prototype)

- **Spring provides many bean scopes like singleton, prototype, request, session, etc.**
- **But here, we will focus on the two most commonly used scopes:**

Singleton Scope (Default)

- **Meaning: Only one object (bean) is created per Spring container.**
- **Shared instance is returned every time it is injected or requested.**
- **Default scope if you don’t specify any scope.**

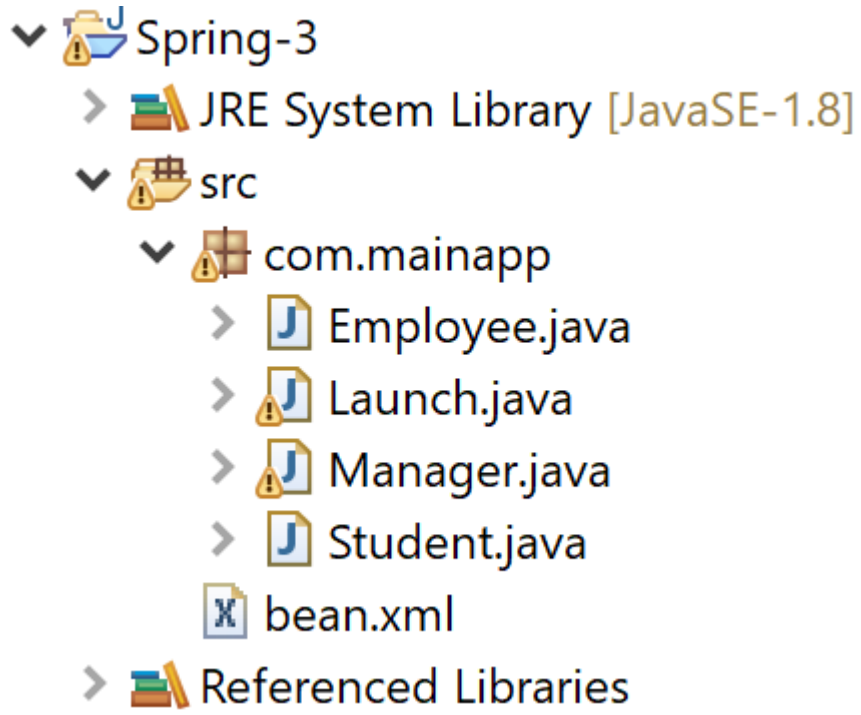
Prototype Scope

- **Meaning: A new object is created every time the bean is requested.**
- **Spring does not manage the full lifecycle beyond creation (no destroy callbacks).**

Note:

In XML configuration, to make the Spring ApplicationContext lazy, you can use the lazy-init="true" attribute in the <bean> tag.

==========Program: lazy-init=true==========




```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->


      <bean id="mng" class="com.mainapp.Manager" >
      </bean>
      <bean id="emp" class="com.mainapp.Employee" lazy-init="true">
      </bean>
      <bean id="std" class="com.mainapp.Student" lazy-init="true" >
      </bean>

</beans>
```

```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {
```

```
	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Manager mng =(Manager) ac.getBean("mng");
		String permission = mng.permission();
		if(permission.equals("GRANTED")) {

			Employee emp =(Employee) ac.getBean("emp");
			Student std =(Student) ac.getBean("std");
			emp.test();
			std.test();
		}
		else {
			System.out.println("YOU ARE NOT AUTHORIZED USER TO
ACCESS EMPLOYEE AND STUDENT");
		}

	}
}
```

```
package com.mainapp;

import java.util.Scanner;

public class Manager {

	static {
		System.out.println("MNG BEAN LOADING");
	}

	public Manager() {
		System.out.println("MNG BEAN INSTANTIATED");
	}

	public String permission() {
		System.out.println("MNG TESTED....");
		Scanner scanner = new Scanner(System.in);
		System.out.println("ENTER KEY");
		int key=scanner.nextInt();

		if(key==1111) {
			return "GRANTED";
		}else {
			return "NOT_GRANTED";
		}

	}
```

```
}
```

```
package com.mainapp;

public class Employee {

	static {
		System.out.println("EMP BEAN LOADING");
	}

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATED");
	}

	public void test() {
		System.out.println("EMP TESTED....");
	}

}
```

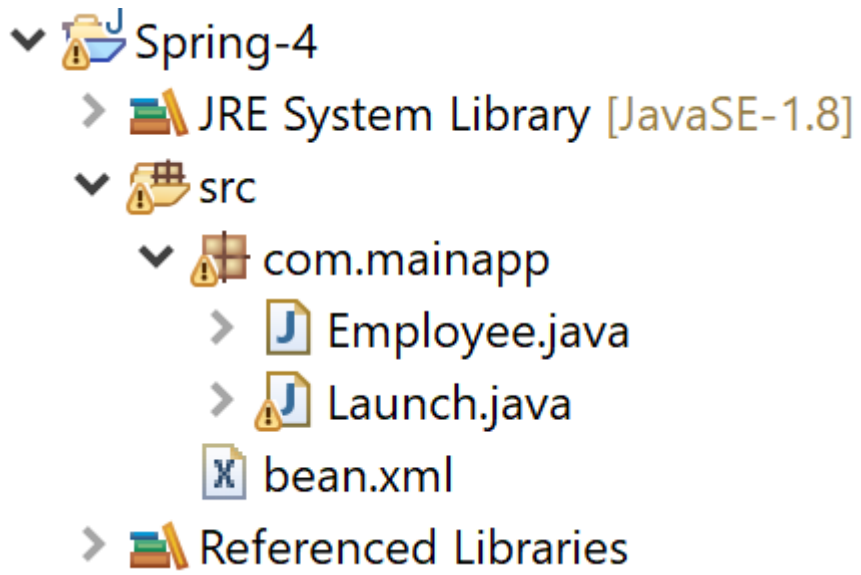
```
package com.mainapp;

public class Student {

	static {
		System.out.println("STD BEAN LOADING");
	}

	public Student() {
		System.out.println("STD BEAN INSTANTIATED");
	}

	public void test() {
		System.out.println("STD TESTED....");
	}

}
```

==========

==========

==========

==========Program: bean scope==========




```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Employee emp1 =(Employee) ac.getBean("emp");
		emp1.setTest(1000);
		emp1.test();

		Employee emp2 =(Employee) ac.getBean("emp");
		emp2.test();
```

```
	}
}
```

```
package com.mainapp;
public class Employee {

	private int test;

	static {
		System.out.println("EMP BEAN LOADING");
	}

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATED");
	}

	public void test() {
		System.out.println("EMP TESTED...."+test);
	}

	public int getTest() {
		return test;
	}

	public void setTest(int test) {
		this.test = test;
	}

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:util="http://www.springframework.org/schema/util"
	xsi:schemaLocation="
	  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
	  http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->

	<bean id="emp" class="com.mainapp.Employee" scope="prototype">
	</bean>

</beans>
```

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

Day-4

Bean Life Cycle

The Bean Life Cycle in Spring refers to the sequence of events that occur from the creation of a bean to its destruction in the Spring container (like ApplicationContext or BeanFactory).

Instantiation

- **Spring creates the bean object using the constructor.**
- **This is done via reflection.**

- Three ways to instantiate a Bean (<bean> , static factory method(CUSTOM LOGIC) , instance method(CUSTOM LOGIC + ADDED DEPEDENCY)

Dependency Injection

- Spring injects values or dependencies into the bean.
- Done via constructor or setter injection.
- Or field injection (with @Autowired, @Value)

Initialization

- Bean performs custom initialization

Bean is Ready to Use

- Fully initialized and managed by Spring.
- Now you can call custom utility methods.

Destruction

- Called when the container shuts down

✅ Bean Instantiation Examples in Spring:

1. Simple Object Creation (No Dependency, No Logic)

- You can use:
- <bean id="emp" class="com.mainapp.Employee"/>
- Spring will use the default constructor — no values, no logic.

2. Inject Values (Dependency Injection Without Custom Logic)

- You can still use <bean> with <property> or <constructor-arg>:

```
<bean id="emp"
class="com.mainapp.Employee">
    <property name="id" value="101"/>
</bean>
```

OR

```
<bean id="emp"
class="com.mainapp.Employee">
    <constructor-arg value="101"/>
</bean>
```

3. Add Custom Logic at Object Creation (e.g., UUID)

- If you want logic like UUID generation or some defaults during creation, then:
- <bean id="emp" class="com.mainapp.ObjectFactory" factory-method="getObject"/>
- ✅ This is the static factory method approach.
- But here Spring cannot inject dependencies like Account inside that method directly — because Spring doesn't manage static methods' internals.

4. Add Logic + Dependency Injection (e.g., UUID (Universally Unique Identifier) + Account from Spring)

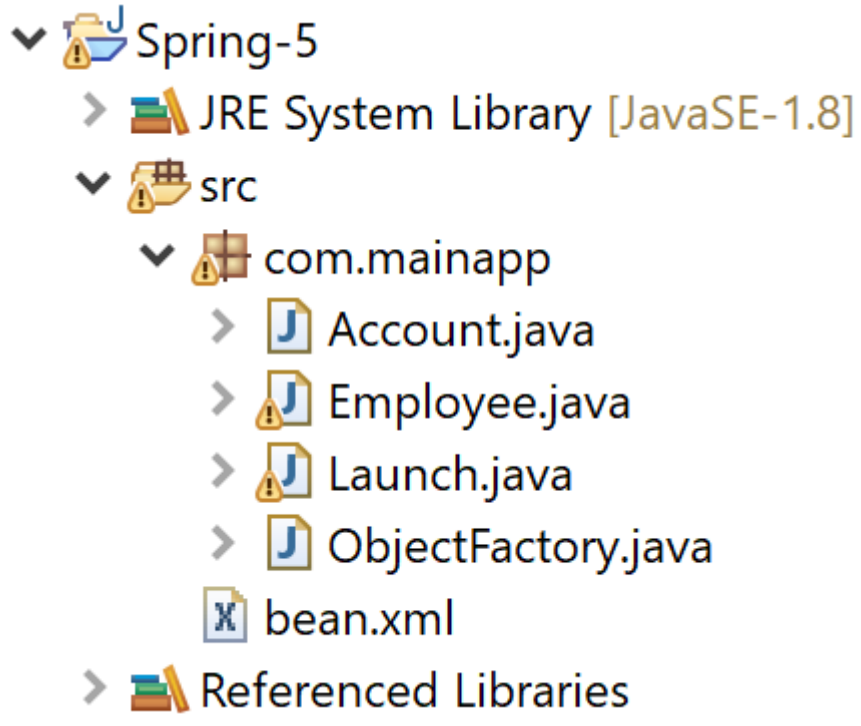
- **Use instance factory method:**

```
<bean id="account"
class="com.mainapp.Account">

    <property name="an"
value="1111"/>

    <property name="bankName"
value="HDFC"/>

</bean>


<bean id="factory"
class="com.mainapp.ObjectFactory">

    <property name="account"
ref="account"/>

</bean>
```

```
   <bean id="employee"
factory-bean="factory"
factory-method="getObject"/>
```

- ✅ **This way you get:**
- **Custom logic (UUID, etc.)**
- **Dependencies from Spring (Account)**

==========Program: bean Life Cycle(Bean Instantiation)==========



bean.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"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->

      <bean id="acc" class="com.mainapp.Account" >
         <property name="an" value="1212"></property>
          <property name="bankName" value="bname"></property>
      </bean>

      <!-- <bean id="emp" class="com.mainapp.ObjectFactory"
factory-method="getObject" >
        <property name="account" ref="acc"  >   </property>
      </bean> -->

      <bean id="objectFactory" class="com.mainapp.ObjectFactory" >
        <property name="account" ref="acc"  >   </property>
      </bean>

      <bean id="emp" factory-bean="objectFactory"
factory-method="getObject" >  </bean>

</beans>
```

```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

      public static void main(String[] args) {

            ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
            Employee emp =(Employee) ac.getBean("emp");
            emp.test();
            System.out.println(emp);

      }
}
```

```java
package com.mainapp;
import java.util.UUID;
public class Employee {

	private String id;  //dependency
	private Account account;  //Spring create & inject

	public void setAccount(Account account) {
		this.account = account;
	}

	public Employee(String id) {
		super();
		System.out.println("EMP BEAN INSTANTIATED THROUGH PC");
		this.id = id;
	}

	public void setId(String id) {
		this.id = id;
	}

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATED");
	}

	public void test() {
		System.out.println("EMP TESTED...."+id);
	}

	@Override
	public String toString() {
		return "Employee [id=" + id + ", account=" + account +
"]";
	}

//	public static Employee getObject() {
//
//		Employee employee = new Employee();
//		employee.setId(UUID.randomUUID().toString());
//		return employee;
//
//	}


}


package com.mainapp;
public class Account {
```

```
	private int an;
	private String bankName;

	public Account() {
		// TODO Auto-generated constructor stub
	}

//	public Account(int an, String bankName) {
//		super();
//		this.an = an;
//		this.bankName = bankName;
//	}

	public int getAn() {
		return an;
	}
	public void setAn(int an) {
		this.an = an;
	}
	public String getBankName() {
		return bankName;
	}
	public void setBankName(String bankName) {
		this.bankName = bankName;
	}
	@Override
	public String toString() {
		return "Account [an=" + an + ", bankName=" + bankName +
"]";
	}

}


package com.mainapp;
import java.util.UUID;

public class ObjectFactory {

	private Account account;

	public void setAccount(Account account) {
		this.account = account;
	}

	public Employee getObject() {

		Employee employee = new Employee();
```

```
            employee.setId(UUID.randomUUID().toString());
            employee.setAccount(account);
            return employee;
        }
}
```

==========

==========

==========

Day-5

What is Bean Initialization?

- **Bean Initialization = The process where Spring creates an object (bean), sets its properties, and executes any custom logic after that.**
- **It happens after the bean is created and its dependencies are injected.**

✅ Real-Time Use Cases of Bean Initialization

1. Loading Initial Data

- When your application starts, you may want to:
- Load some default records into the database
- Read from a file or external API

2. Generating Tokens / Keys

- If your app requires some security keys, UUIDs, or tokens to be generated at startup.

3. Validating Config or Environment

- Check whether required configuration values are set properly at startup.

What is Bean Destruction?

- Bean destruction is the process where Spring cleans up resources used by a bean before removing it from the container (usually during application shutdown).
- It's like a “goodbye method” where you release memory, close connections, or stop background tasks.

- **Works only for singleton-scoped beans (not for prototype by default).**

✅ Real-Time Use Cases of Bean Destruction

1. Closing Database or Network Connections

You may have opened JDBC, Redis, or Mongo connections.

2. Closing File Resources / Streams

If you're writing logs or files, make sure to close file writers.

3. Sending Shutdown Alerts / Logs

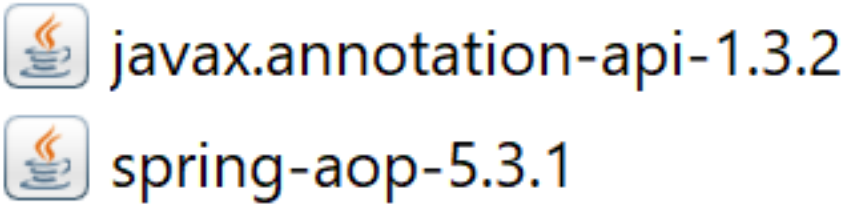
Notify your monitoring system or log service on shutdown.

✅ There are 3 actual ways to define init and destroy logic:

1. Annotations
2. CallBack Interfaces
3. XML

Annotations (@PostConstruct, @PreDestroy)

- Modern way, recommended in most cases.
- Very clean and quick, no need for separate XML or interface implementation.
- You need to include two extra jars



Callback Interfaces

- **Spring provides interfaces to hook into the bean lifecycle:**
- **InitializingBean → method: afterPropertiesSet()**
- **DisposableBean → method: destroy()**

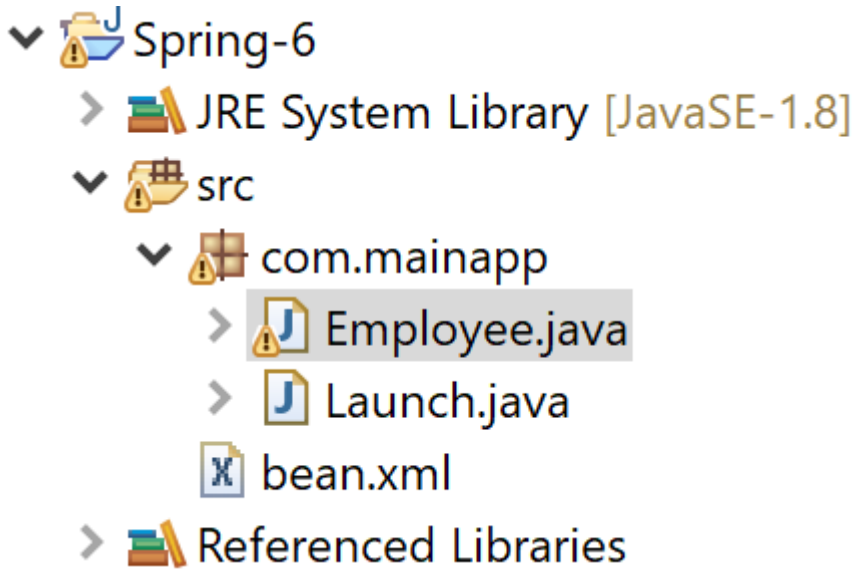
✅ **Use When:**

- ❖ **You want more control with structured lifecycle methods.**
- ❖ **Often used in older/legacy projects.**

XML or @Bean(initMethod = ...)

- **Used when you can't modify the class — like third-party libraries.**
- **You define which method to call during init/destroy externally (in XML or Java config)**

==========Program: bean Life Cycle(Bean Initialization)==========



bean.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"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->


      <!-- <bean id="emp" class="com.mainapp.Employee"
init-method="xmlInit" destroy-method="xmlDestroy" >  </bean> -->

    <bean id="emp" class="com.mainapp.Employee" scope="prototype">
</bean>
```

```
  <!--   <bean
class="org.springframework.context.annotation.CommonAnnotationBeanPo
stProcessor" ></bean> -->

</beans>
```

```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Employee emp =(Employee) ac.getBean("emp");
		emp.test();
		try {
			emp.destroy();
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		ac.close();

	}
}
```

```
package com.mainapp;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Employee implements InitializingBean, DisposableBean {

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATION");
	}

//	//initialization
//	public void xmlInit() {
//		System.out.println("RESORUCE ALLOCATION");
//	}
```

```
//    @PostConstruct
//    public void annoInit() {
//          System.out.println("RESORUCE ALLOCATION");
//    }

      @Override
      public void afterPropertiesSet() throws Exception {
            System.out.println("RESORUCE ALLOCATION");
      }

      public void test() {
            System.out.println("EMP BEAN TESTED....");
      }

//     //destroy
//    public void xmlDestroy() {
//          System.out.println("RESORUCE De-ALLOCATION");
//    }

//    @PreDestroy
//    public void annoDestroy() {
//          System.out.println("RESORUCE De-ALLOCATION");
//    }

      @Override
      public void destroy() throws Exception {
            System.out.println("RESORUCE De-ALLOCATION");
      }

}
```

==========

==========

==========

Day-6

What is BeanPostProcessor?

BeanPostProcessor is an interface in the Spring Framework that allows you to intercept bean creation after the initialization and before the bean is used. You can use it to perform custom logic on beans — like modifying, wrapping, or injecting extra functionality.

How It Works Internally:

- **If you define a BeanPostProcessor, Spring will automatically invoke:**
    1. **postProcessBeforeInitialization(bean, beanName)**
    2. **postProcessAfterInitialization(bean, beanName)**
- **for every single bean that is created in the ApplicationContext.**

Lifecycle Hook Points

- **postProcessBeforeInitialization(...)**
  - **Called before any bean's init-method.**
- **postProcessAfterInitialization(...)**
  - **Called after bean initialization is complete.**

Here are real-time, practical uses of BeanPostProcessor in real Spring projects:

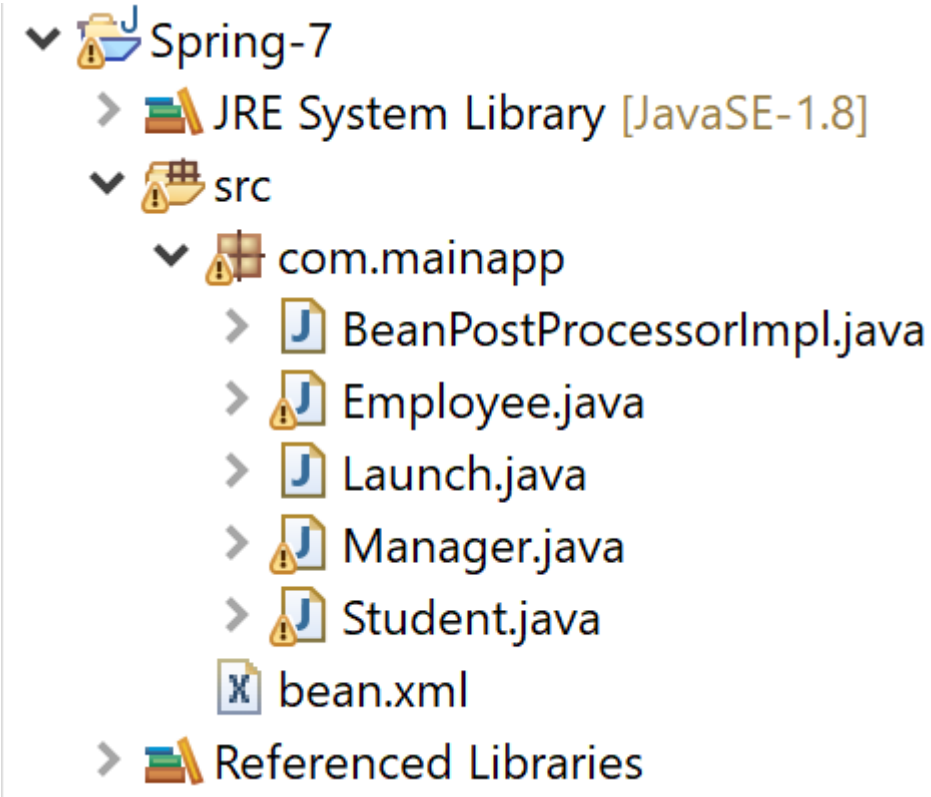
Custom Logger Injection

- **Imagine you want to inject a logger in all beans automatically (without writing code in each class):**

Encrypt/Decrypt Bean Properties

- **For example, you store encrypted passwords in application properties. You want to decrypt them before the bean is used.**

==========Program: BeanPostProcessor==========



bean.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"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->

    <bean id="employee" class="com.mainapp.Employee">  </bean>
    <bean id="manager"  class="com.mainapp.Manager">  </bean>
    <bean id="student"  class="com.mainapp.Student">  </bean>
    <bean class="com.mainapp.BeanPostProcessorImpl" ></bean>

    <bean
class="org.springframework.context.annotation.CommonAnnotationBeanPo
stProcessor" ></bean>


</beans>
```

```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

      public static void main(String[] args) {

            ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
            ac.close();

      }
}
```

```
package com.mainapp;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
public class Employee {

      public Employee() {
            System.out.println("EMP BEAN INSTANTIATION");
      }

      //Employee Object and Employee Bean Name

      @PostConstruct
      public void init() {
            System.out.println("EMP BEAN INITIALIZATION");
      }
```

```
	public void test() {
		System.out.println("EMP BEAN CUSTOM METHOD....");
	}


	@PreDestroy
	public void annoDestroy() {
		System.out.println("EMP BEAN DETSROY");
	}
}
```

```
package com.mainapp;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Manager {

	public Manager() {
		System.out.println("MANAGER BEAN INSTANTIATION");
	}

	@PostConstruct
	public void init() {
		System.out.println("MANAGER BEAN INITIALIZATION");
	}


	public void test() {
		System.out.println("MANAGER BEAN CUSTOM METHOD....");
	}


	@PreDestroy
	public void annoDestroy() {
		System.out.println("MANAGER BEAN DETSROY");
	}

}
```

```
package com.mainapp;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
```

```java
public class Student {

	public Student() {
		System.out.println("STUDENT BEAN INSTANTIATION");
	}

	@PostConstruct
	public void init() {
		System.out.println("STUDENT BEAN INITIALIZATION");
	}


	public void test() {
		System.out.println("STUDENT BEAN CUSTOM METHOD....");
	}

	@PreDestroy
	public void annoDestroy() {
		System.out.println("STUDENT BEAN DETSROY");
	}

}
```

```java
package com.mainapp;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class BeanPostProcessorImpl implements BeanPostProcessor{

	@Override
	public Object postProcessBeforeInitialization(Object bean,
String beanName) throws BeansException {

		System.out.println("postProcessBeforeInitialization");

		if(bean instanceof  Employee) {
			System.out.println("EMPLOYEE SPECIFIC WORK....Ex
PASSOWORD DECRYPT");
		}
		else if(bean instanceof Manager) {
			Manager manger=(Manager) bean;
			System.out.println("BEAN IS PROPERTY INSTANTIATED
AND DEPENENCIES ARE INJECTED FOR THE BEAN "+beanName);
			System.out.println(manger);
		}
		return bean;
	}

	@Override
```

```
     public Object postProcessAfterInitialization(Object bean,
String beanName) throws BeansException {
          System.out.println("postProcessAfterInitialization");

          if(bean instanceof  Employee) {
               System.out.println("EMPLOYEE VALIDATED
SUCCESSFULLY");
          }
          else if(bean instanceof Manager) {
               System.out.println("MANAGER CONNECTION POOL
MODIFIED");
          }

          return bean;
     }
}
```

==========

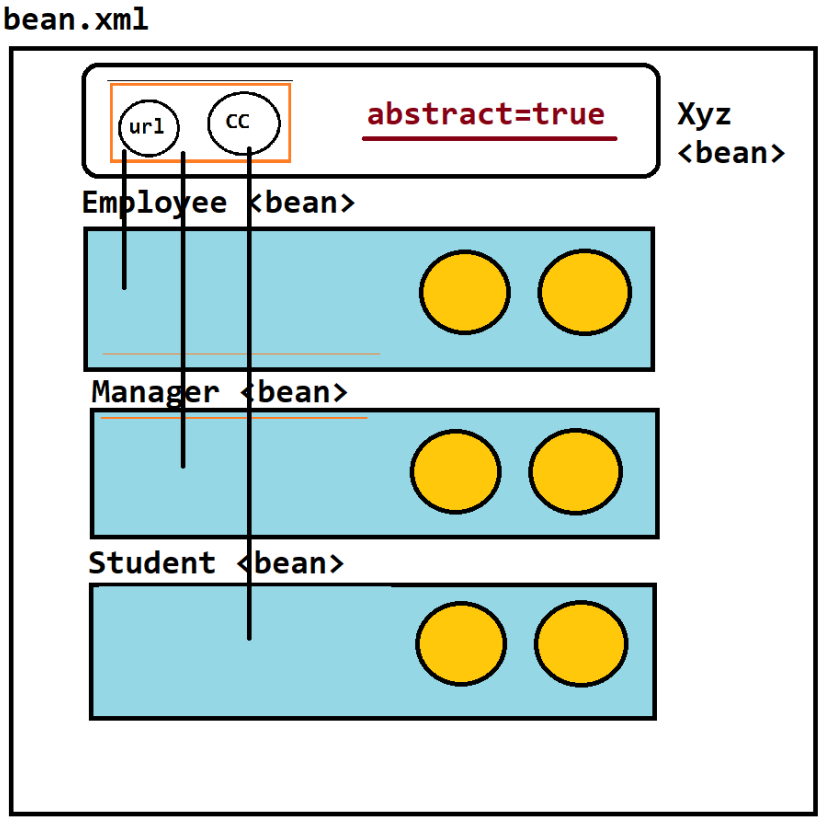
==========

==========

Day-7

What is Bean Inheritance?

Bean inheritance in Spring allows one bean definition to inherit configuration data from another bean. This is useful when you have multiple beans with common properties, and you want to avoid repetition.



✅ Syntax Example (XML-based)

```
<bean id="parentBean" class="com.example.Person">
    <property name="name" value="Default Name" />
    <property name="age" value="30" />
```

```
</bean>


<bean id="childBean" class="com.example.Employee"
parent="parentBean" >
   <property name="designation" value="Developer"
/>
</bean>
```

✅ How it works

- childBean inherits the name and age properties from parentBean.
- It can override the inherited properties if needed.
- The parent bean does not need to be instantiable, i.e., it can be abstract.

✅ Making Parent Bean Abstract (Optional but Recommended)

```
<bean id="parentBean" class="com.example.Person" abstract="true">
    <property name="name" value="Default Name" />
    <property name="age" value="30" />
</bean>
```

Setting abstract="true" ensures that parentBean is only used for inheritance and cannot be instantiated directly.

✅ Use Cases

- Reusing common properties (like database configs, common field values).
- Defining template-like base beans.
- Cleaner and DRY (Don't Repeat Yourself) configurations.

✅ Important Notes

- **Works only in XML configuration.**
- **Not supported in Java-based (@Configuration) or annotation-based bean definitions.**
- **Child can override or add its own property values.**

Dependency Injection (DI)

- **Dependency Injection (DI) in Spring is a technique to provide objects that a class needs (its dependencies) rather than creating them directly.**
- **Spring manages the object lifecycle and dependencies — this is part of the Inversion of Control (IoC) principle.**

✅ Benefits of DI in Spring

- Reduces tight coupling
- Easier to test and maintain
- Promotes code reusability and modular design

Types of Dependency Injection in Spring

Type	Description
Constructor Injection	Dependencies are provided through the constructor
Setter Injection	Dependencies are set using setter methods
Field Injection	Dependencies are injected directly into fields (Not Supported in XML)

Let's start learning Dependency Injection using the XML-based approach first.

✅ Dependency Injection using XML Approach

- **In the XML-based configuration, only Setter Injection and Constructor Injection are supported for dependency injection in Spring.**

1. Setter Injection – Uses <property> tag

✅ **Example values:**

- Primitive or String → value attribute
- User-defined object → ref attribute

```
<property name="propertyName"
value="someValue"/>     <!-- Primitive or
String -->

<property name="propertyName"
ref="beanId"/>          <!-- Reference to
another bean -->
```

2. Constructor Injection – Uses <constructor-arg> tag

✅ **Example values:**

- Primitive or String → value attribute
- User-defined object → ref attribute

```
<constructor-arg  name="propertyName"
value="someValue"/>        <!-- Primitive or String
-->

<constructor-arg  name="propertyName"
ref="beanId"/>                <!-- Reference to
another bean -->
```

What is Collection Injection?

- **Collection injection is the process of injecting collections of values or objects into a Spring bean using XML configuration.**
- **This is useful when a bean has fields like List<String>, Set<String>, Map<String, Integer>, or Properties, and you want Spring to provide those values automatically.**

What is List Injection?

- **List Injection is the process of injecting a group of ordered values or objects into a bean property of type List<T> using XML in Spring.**
- **Spring supports two main ways of injecting a List:**
    1. ✅ **Through Setter Injection**
    2. ✅ **Through Constructor Injection**

✅ List Collection Injection Through Setter Injection

```
<bean id="student"
class="com.codehunt.Student">
    <property name="subjects">
        <list>
            <value>Math</value>
            <value>Science</value>
            <value>English</value>
        </list>
    </property>
</bean>
```

- This injects a list of strings into the subjects property using the setter method.
- By default, Spring uses java.util.ArrayList to inject the list unless you specify another implementation.
- So when you use <list>, Spring automatically wraps the items into an ArrayList.
- In case of User Defined Object like Account you should use <ref> tag inside <list>

Use Your Own List Implementation Standalone List (Modularity & Reuse)

- **If you want to reuse the same list in multiple beans:**

```
<util:list id="subjectList">
    <value>Hibernate</value>
    <value>JPA</value>
</util:list>


<bean id="student"
class="com.codehunt.Student">
    <property name="subjects"
ref="subjectList"/>
</bean>
```

- **This keeps your config modular and reusable.**
- **In util:list, we can specify the list implementation using the list-class attribute. So you're not limited to ArrayList only.**

- In case of User Defined Object like Account you should use <ref> tag inside <util:list>

```
<util:list id="subjectList"
list-class="java.util.LinkedList">
    <value>Hibernate</value>
    <value>JPA</value>
</util:list>
```

Separate Bean Tag (Custom Collection Injection)

- The <bean class="..."> approach is designed to define any kind of object, not just collections. While util:list is specifically meant for creating and injecting collections like List, the <bean> tag is much more flexible. It allows you to create full-fledged Java objects—whether it's a collection, a custom class, a service, or any POJO. This makes it suitable not only for injecting collections but also for wiring complex beans that

may have multiple properties, dependencies, or lifecycle methods.

- In case of User Defined Object like Account you should use <ref> tag inside <list>

```
<bean id="list"
class="java.util.LinkedList">
    <constructor-arg>
        <list>
            <value>JAVA</value>
            <value>PYTHON</value>
            <value>PHP</value>
        </list>
    </constructor-arg>
</bean>

<bean id="employee"
class="com.test.Employee">
  <property name="deafultListOfBooks"
ref="list"/>
</bean>
```

✅ List Collection Injection Through Constructor Injection

- **Injecting List via Constructor**

```
<bean id="student"
class="com.codehunt.Student">
    <constructor-arg name="subjects">
        <list>
            <value>Math</value>
            <value>Science</value>
            <value>English</value>
        </list>
    </constructor-arg>
</bean>
```

- **This injects a list of strings into the subjects parameter using the constructor.**
- **By default, Spring uses java.util.ArrayList unless you explicitly specify another implementation.**

- So when you use <list> inside constructor-arg, Spring automatically wraps the values into an ArrayList.
- In case of User Defined Object like Account you should use <ref> tag inside <list>

Use Your Own List Implementation: Standalone List (Modularity & Reuse)

- Use Standalone List for Modularity & Reuse
- If the same list is used in multiple places, define it as a standalone util bean:

```
<util:list id="subjectList">
    <value>Hibernate</value>
    <value>JPA</value>
</util:list>
<bean id="student"
class="com.codehunt.Student">
```

```
    <constructor-arg name="subjects"
ref="subjectList"/>

</bean>
```

- This approach makes your configuration modular and reusable.
- You can also specify the list implementation using the list-class attribute:
- In case of User Defined Object like Account you should use <ref> tag inside <util:list>

```
<util:list id="subjectList"
list-class="java.util.Vector">

    <value>Hibernate</value>

    <value>JPA</value>

</util:list>
```

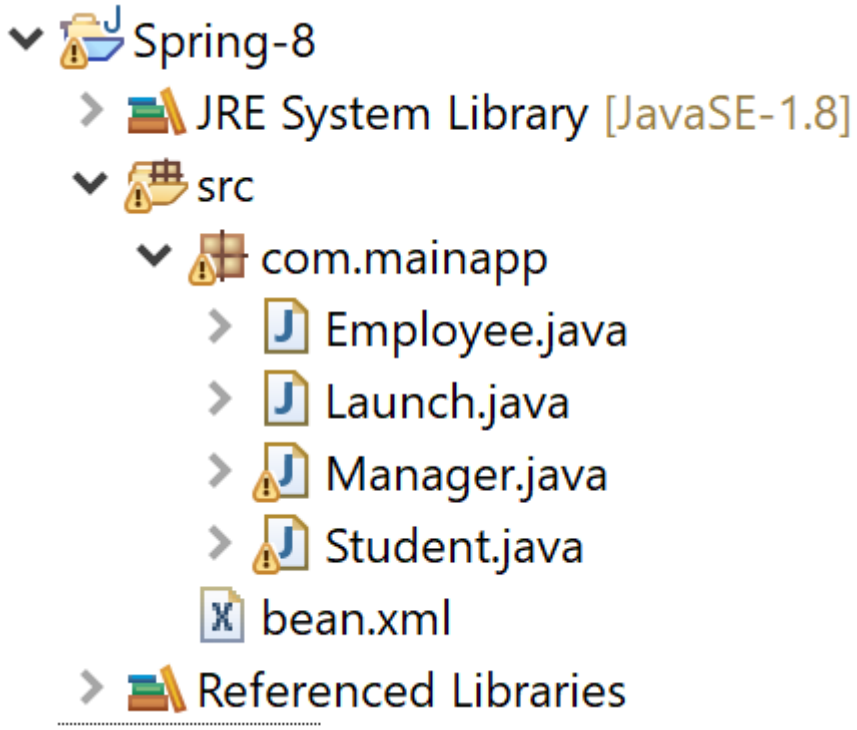
Separate Bean Tag (Custom Collection Injection)

- Instead of using util:list, you can also create a collection bean using a <bean> tag. This gives you full flexibility.

```
<bean id="list"
class="java.util.LinkedList">
    <constructor-arg>
        <list>
            <value>JAVA</value>
            <value>PYTHON</value>
            <value>PHP</value>
        </list>
    </constructor-arg>
</bean>
<bean id="employee"
class="com.test.Employee">
  <constructor-arg name="deafultListOfBooks"
ref="list"/>
</bean>
```

- The <bean class="..."> tag is not limited to collections. You can define any kind of object — collections, POJOs, or services.
- It supports full lifecycle methods, dependencies, and property injections.
- In case of User Defined Object like Account you should use <ref> tag inside <list>

==========Program: Bean Inheritance==========




```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Employee employee=  (Employee) ac.getBean("employee");
		Manager manager=  (Manager) ac.getBean("manager");
		Student student=  (Student) ac.getBean("student");

		System.out.println(employee);
		System.out.println(manager);
```

```
            System.out.println(student);

            ac.close();

      }
}
package com.mainapp;
public class Employee {

      private String url;
      private String countryCode;
      private int eid;

      public Employee() {
            System.out.println("EMP BEAN INSTANTIATION");
      }

      public void setUrl(String url) {
            this.url = url;
      }

      public void setCountryCode(String countryCode) {
            this.countryCode = countryCode;
      }

      public void setEid(int eid) {
            this.eid = eid;
      }

      @Override
      public String toString() {
            return "Employee [url=" + url + ", countryCode=" +
countryCode + ", eid=" + eid + "]";
      }
}



package com.mainapp;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
public class Manager {

      private String url;
      private String countryCode;
      private int mid;

      public Manager() {
```

```
		System.out.println("MANAGER BEAN INSTANTIATION");
	}


	public void setUrl(String url) {
		this.url = url;
	}

	public void setCountryCode(String countryCode) {
		this.countryCode = countryCode;
	}

	public void setMid(int mid) {
		this.mid = mid;
	}

	@Override
	public String toString() {
		return "Manager [url=" + url + ", countryCode=" +
countryCode + ", mid=" + mid + "]";
	}

}



package com.mainapp;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
public class Student {

	private String url;
	private String countryCode;
	private int roll;

	public Student() {
		System.out.println("STUDENT BEAN INSTANTIATION");
	}

	public void setUrl(String url) {
		this.url = url;
	}

	public void setCountryCode(String countryCode) {
		this.countryCode = countryCode;
	}

	public void setRoll(int roll) {
```

```
		this.roll = roll;
	}

	@Override
	public String toString() {
		return "Student [url=" + url + ", countryCode=" +
countryCode + ", roll=" + roll + "]";
	}

}
```

bean.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"
	xmlns:util="http://www.springframework.org/schema/util"
	xsi:schemaLocation="
	  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
	  http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->

    <bean id="parentBean" abstract="true" >
	  <property name="url"
value="jdbc:mysql://localhost:3306"></property>
	  <property name="countryCode"  value="1122"></property>
    </bean>

    <bean id="employee" class="com.mainapp.Employee"
parent="parentBean" >
	  <property name="eid"  value="123412"></property>
    </bean>

    <bean id="manager"  class="com.mainapp.Manager"
parent="parentBean">
	  <property name="mid"  value="345678"></property>
    </bean>

    <bean id="student"  class="com.mainapp.Student"
parent="parentBean">
	  <property name="url"
value="jdbc:mysql://localhost:3308"></property> <!--WE CAN OVERRIDE
-->
	  <property name="roll"  value="12"></property>
```

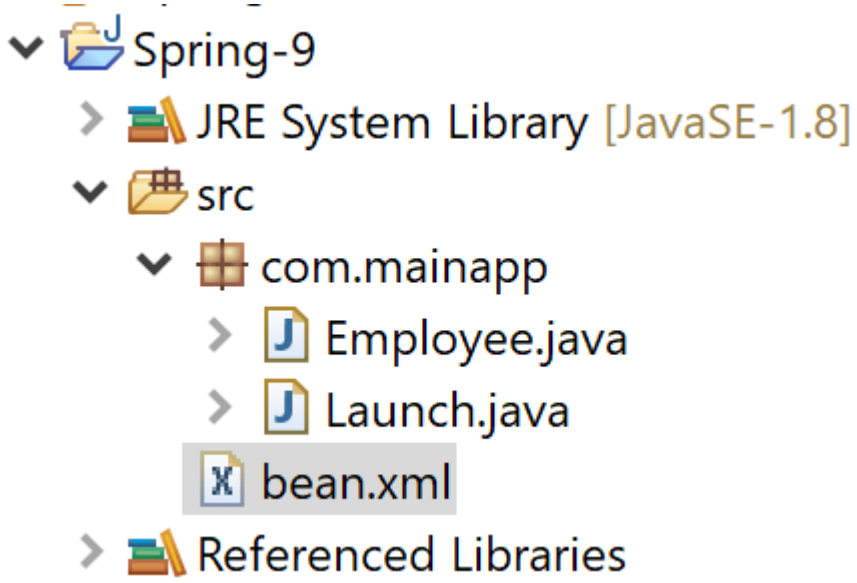
```
    </bean>

</beans>
```

==========

==========

==========

==========Program: Dependency Injection(List)==========




```
package com.mainapp;
```

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Employee employee=  (Employee) ac.getBean("employee");
		System.out.println(employee.getListOfBooks().getClass());
		System.out.println(employee);
		ac.close();

	}
}


package com.mainapp;
import java.util.List;
public class Employee {

	private int eid;
	private String ename;
	private List<String> listOfBooks;  //  private List<Account>
listOfAccount;

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATION");
	}

	public void setEid(int eid) {
		this.eid = eid;
	}

	public void setEname(String ename) {
		this.ename = ename;
	}

	public void setListOfBooks(List<String> listOfBooks) {
		this.listOfBooks = listOfBooks;
	}

	public int getEid() {
		return eid;
	}

	public String getEname() {
		return ename;
	}
```

```
	public List<String> getListOfBooks() {
		return listOfBooks;
	}

	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" + ename + ",
listOfBooks=" + listOfBooks + "]";
	}
}
```

bean.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"
	xmlns:util="http://www.springframework.org/schema/util"
	xsi:schemaLocation="
	  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
	  http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd"> <!--
bean definitions here -->


	<bean id="employee" class="com.mainapp.Employee" >
	  <property name="eid"  value="123412"></property>
	  <property name="ename"  value="raju"></property>
	  <property name="listOfBooks">
	    <list >
	      <value>JAVABYCODEHUNT</value>
	      <value>SPRINGBYCODEHUNT</value>
	      <value>JAVAEEBYCODEHUNT</value>
	    </list>
	  </property>
	</bean>


	<!--  <bean id="employee" class="com.mainapp.Employee" >
	  <property name="eid"  value="123412"></property>
	  <property name="ename"  value="raju"></property>
	  <property name="listOfBooks" ref="myList" > </property>
	</bean>
```

```
 STANDALONE LIST
 <util:list id="myList" list-class="java.util.LinkedList" >
     <value>JAVABYCODEHUNT</value>
     <value>SPRINGBYCODEHUNT</value>
     <value>JAVAEEBYCODEHUNT</value>
 </util:list> -->


  <!-- <bean id="employee" class="com.mainapp.Employee" >
     <property name="eid"  value="123412"></property>
     <property name="ename"  value="raju"></property>
     <property name="listOfBooks" ref="myList" > </property>
 </bean>


 <bean id="myList" class="java.util.LinkedList" >

    <constructor-arg>
       <list>
          <value>JAVABYCODEHUNT</value>
          <value>SPRINGBYCODEHUNT</value>
          <value>JAVAEEBYCODEHUNT</value>
       </list>
    </constructor-arg>

 </bean> -->



<!--  <bean id="acc1" class="com.mainapp.Account" > </bean>
 <bean id="acc2" class="com.mainapp.Account" > </bean>
 <bean id="acc3" class="com.mainapp.Account" > </bean>

 <bean id="employee" class="com.mainapp.Employee" >
     <property name="eid"  value="123412"></property>
     <property name="ename"  value="raju"></property>
     <property name="listOfAccount">
        <list>
          <ref bean="acc1"  />
          <ref bean="acc1"  />
          <ref bean="acc1"  />
        </list>
     </property>
 </bean>  -->



<!--  <bean id="acc1" class="com.mainapp.Account" > </bean>
```

```
    <bean id="acc2" class="com.mainapp.Account" > </bean>
    <bean id="acc3" class="com.mainapp.Account" > </bean>

    <util:list id="myList" list-class="java.util.LinkedList" >
        <ref bean="acc1"  />
        <ref bean="acc1"  />
        <ref bean="acc1"  />
    </util:list>

     <bean id="employee" class="com.mainapp.Employee" >
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="listOfAccount" ref="myList" > </property>
    </bean> -->



    <bean id="acc1" class="com.mainapp.Account" > </bean>
    <bean id="acc2" class="com.mainapp.Account" > </bean>
    <bean id="acc3" class="com.mainapp.Account" > </bean>

     <bean id="employee" class="com.mainapp.Employee" >
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="listOfAccount" ref="myList" > </property>
    </bean>


    <bean id="myList" class="java.util.LinkedList" >

       <constructor-arg>
          <list>
             <ref bean="acc1"  />
             <ref bean="acc1"  />
             <ref bean="acc1"  />
          </list>
       </constructor-arg>

    </bean>


</beans>
```

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

Day-8

✅ Dependency Injection using XML Approach

What is Set Injection?

Set Injection is the process of injecting a collection of unique, unordered values or objects into a bean property of type Set<T> using Spring XML configuration.

Spring supports two main approaches to inject a Set:

1. ✅ **Through Setter Injection**
2. ✅ **Through Constructor Injection**

✅ Set Collection Injection Through Setter Injection

```
<bean id="student"
class="com.codehunt.Student">
    <property name="skills">
        <set>
            <value>Java</value>
            <value>Python</value>
            <value>SQL</value>
        </set>
    </property>
</bean>
```

- **This injects a set of strings into the skills property using the setter method.**

- **By default, Spring uses java.util.LinkedHashSet for <set> unless specified otherwise.**
- **In case of User Defined Objects (like Book, Account, etc.), use <ref> tags inside <set>:**

```
<set>
    <ref bean="book1"/>
    <ref bean="book2"/>
</set>
```

✅ Reusable Set with <util:set> (Modular & Reusable Configuration)

```
<util:set id="skillSet">
    <value>Spring</value>
    <value>Hibernate</value>
</util:set>
```

```
<bean id="developer"
class="com.codehunt.Developer">
    <property name="skills"
ref="skillSet"/>
</bean>
```

- This allows you to reuse the same set in multiple beans.
- You can also define the set implementation using the set-class attribute.

```
<util:set id="techSet"
set-class="java.util.TreeSet">
    <value>Docker</value>
    <value>Kubernetes</value>
</util:set>
```

- For User Defined Objects, again use <ref> inside <util:set>:

```
<util:set id="bookSet">
    <ref bean="book1"/>
    <ref bean="book2"/>
</util:set>
```

✅ Set Collection Injection Using Seperate <bean> Tag

- **The <bean> tag can be used to create a Set object manually and then inject it into another bean.**

```
<bean id="bookSet" class="java.util.HashSet">
    <constructor-arg>
        <set>
            <value>Core Java</value>
            <value>DSA</value>
            <value>System Design</value>
        </set>
    </constructor-arg>
</bean>

<bean id="library" class="com.codehunt.Library">
```

```
    <property name="books"
ref="bookSet"/>

</bean>
```

- **This approach offers full control and flexibility to construct any Java object, including collections.**
- **Ideal for injecting complex sets or custom configuration.**

✅ Set Injection Through Constructor

• In Spring, you can inject a Set into a bean using constructor injection by placing the <set> element inside a <constructor-arg> tag.

What is Map Injection?

- **Map Injection is the process of injecting key-value pairs into a bean property of type Map<K, V> using Spring XML configuration.**
- **It allows injection of both simple types and complex objects as keys and values.**
- **Spring supports two main ways to inject a Map:**

1. ✅ **Through Setter Injection**
2. ✅ **Through Constructor Injection**

✅ Map Collection Injection Through Setter Injection

```
<bean id="student"
class="com.codehunt.Student">
    <property name="grades">
        <map>
            <entry key="Math" value="A"/>
            <entry key="Science"
value="B"/>
```

```
            <entry key="English"
value="A"/>
        </map>
    </property>
</bean>
```

- This injects a Map<String, String> into the grades property using setter injection.
- By default, Spring uses java.util.LinkedHashMap unless otherwise specified.
- For User Defined Objects as key use key-ref.
- For User Defined Objects as value use value-ref.

```
<map>
  <entry key="account1" value-ref="accBean1"/>
  <entry key="account2" value-ref="accBean2"/>
</map>
```

✅ Reusable Map with <util:map> (Modular & Reusable Configuration)

```
<util:map id="configMap">
    <entry key="timeout" value="30"/>
    <entry key="maxUsers" value="100"/>
</util:map>

<bean id="server"
class="com.codehunt.Server">
    <property name="config"
ref="configMap"/>
</bean>
```

- **Use this approach if the same map is to be used across multiple beans.**
- **You can specify the map implementation using map-class.**

```
<util:map id="envMap"
map-class="java.util.TreeMap">
    <entry key="dev" value="localhost"/>
    <entry key="prod"
value="prod.server.com"/>
</util:map>
```

- **For User Defined Objects, use value-ref and key-ref:**

```
<util:map id="accountMap">
    <entry key="admin"
value-ref="adminAccount"/>
    <entry key="user"
value-ref="userAccount"/>
</util:map>
```

✅ Map Injection Using Seperate <bean> Tag

```
<bean id="subjectMap"
class="java.util.HashMap">
    <constructor-arg>
        <map>
            <entry key="DSA"
value="Completed"/>
            <entry key="Java"
value="Ongoing"/>
        </map>
    </constructor-arg>
</bean>

<bean id="course"
class="com.codehunt.Course">
    <property name="subjectStatus"
ref="subjectMap"/>
</bean>
```

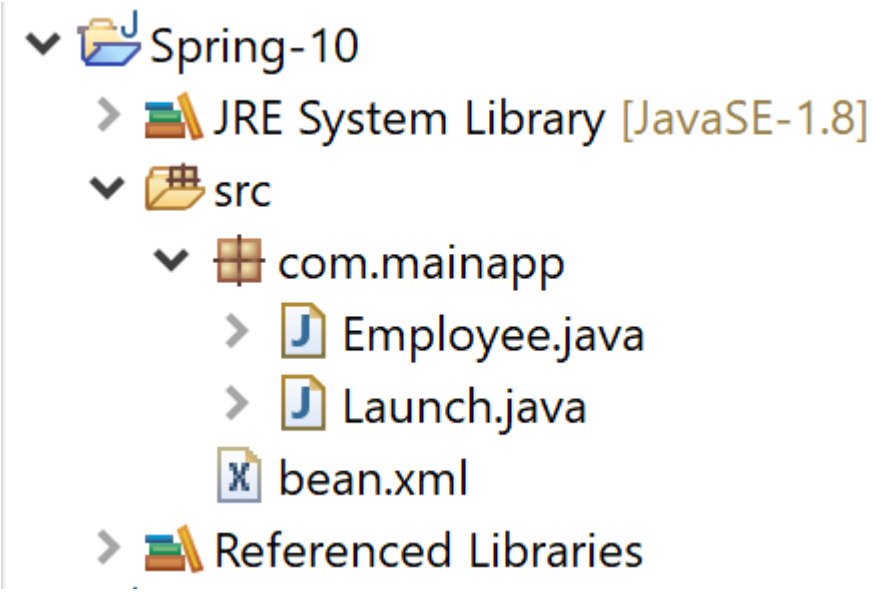
- **This is a fully customizable way to create any type of Map bean manually and inject it.**
- **Works well when creating reusable, complex, or pre-populated maps.**

- **It provides more features than Standalone map**

✅ Map Injection Through Constructor

- **In Spring, you can inject a Map into a bean using constructor injection by passing the <map> element inside a <constructor-arg> tag.**

==========Program: Set injection==========



```java
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Employee employee=  (Employee)
ac.getBean("employee");

System.out.println(employee.getSetOfBooks().getClass());
		System.out.println(employee);
		ac.close();
	}
}
```

```java
package com.mainapp;
import java.util.Set;
public class Employee {

	private int eid;
	private String ename;
	private Set<String> setOfBooks;  //  private
Set<Account> setOfAccount;

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATION");
	}

	public void setEid(int eid) {
		this.eid = eid;
	}

	public void setEname(String ename) {
		this.ename = ename;
	}
```

```
	public int getEid() {
		return eid;
	}

	public String getEname() {
		return ename;
	}


	public Set<String> getSetOfBooks() {
		return setOfBooks;
	}

	public void setSetOfBooks(Set<String> setOfBooks) {
		this.setOfBooks = setOfBooks;
	}

	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" +
ename + ", setOfBooks=" + setOfBooks + "]";
	}
}
```

bean.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"

xmlns:util="http://www.springframework.org/schema/util"
	xsi:schemaLocation="
		http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.x
sd
		http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd
"> <!-- bean definitions here -->
```

```
    <!--
        <bean id="employee" class="com.mainapp.Employee" >
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="setOfBooks">
           <set >
             <value>JAVABYCODEHUNT</value>
             <value>SPRINGBYCODEHUNT</value>
             <value>JAVAEEBYCODEHUNT</value>
           </set>
        </property>
    </bean>   -->


   <!--  <bean id="employee" class="com.mainapp.Employee"
>
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="setOfBooks" ref="mySet" >
</property>
    </bean>


     STANDALONE SET
     <util:set id="mySet" set-class="java.util.TreeSet"  >
        <value>JAVABYCODEHUNT</value>
        <value>SPRINGBYCODEHUNT</value>
        <value>JAVAEEBYCODEHUNT</value>
    </util:set>   -->


    <!-- <bean id="employee" class="com.mainapp.Employee"
>
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="setOfBooks" ref="mySet" >
</property>
    </bean>


    <bean id="mySet" class="java.util.TreeSet" >
```

```
        <constructor-arg>
           <set>
              <value>JAVABYCODEHUNT</value>
              <value>SPRINGBYCODEHUNT</value>
              <value>JAVAEEBYCODEHUNT</value>
           </set>
        </constructor-arg>

    </bean>  -->

</beans>
```

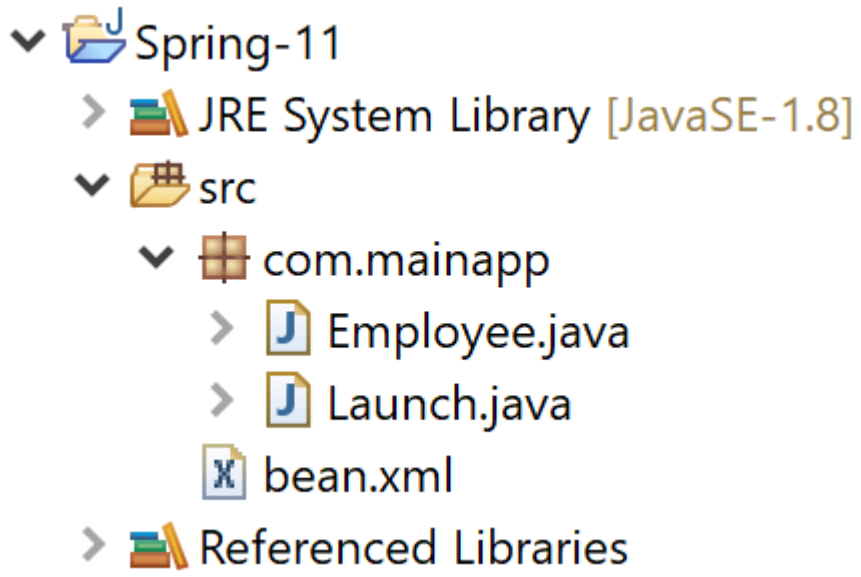
==========

==========

==========

==========Program: Map injection==========

==========




```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Employee employee=  (Employee)
ac.getBean("employee");

System.out.println(employee.getMapOfBooks().getClass());
		System.out.println(employee);
		ac.close();

	}
}
```

```
package com.mainapp;
import java.util.Map;
public class Employee {
```

```
	private int eid;
	private String ename;
	private Map<String,String> mapOfBooks;  //  private
Map<String,Account> mapOfAccount;

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATION");
	}

	public void setEid(int eid) {
		this.eid = eid;
	}

	public void setEname(String ename) {
		this.ename = ename;
	}

	public int getEid() {
		return eid;
	}

	public String getEname() {
		return ename;
	}

	public Map<String, String> getMapOfBooks() {
		return mapOfBooks;
	}

	public void setMapOfBooks(Map<String, String>
mapOfBooks) {
		this.mapOfBooks = mapOfBooks;
	}

	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" +
ename + ", mapOfBooks=" + mapOfBooks + "]";
	}
}
```

bean.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"

xmlns:util="http://www.springframework.org/schema/util"
     xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.x
sd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd
"> <!-- bean definitions here -->

   <!-- <bean id="acc1" class="com.mainapp.Account"
></bean> -->
   <!-- <bean id="std1" class="com.mainapp.Student"
></bean> -->

    <!-- <bean id="employee" class="com.mainapp.Employee"
>
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="mapOfBooks">
           <map >
              <entry key="book1010"
value-ref="acc1"></entry>
              <entry key-ref="std1"
value-ref="acc1"></entry>

              <entry key="book1010"
value="JAVABYCODEHUNT"></entry>
              <entry key="book1011"
value="SPRINGBYCODEHUNT"></entry>
              <entry key="book1012"
value="JAVAEEBYCODEHUNT"></entry>
           </map>
        </property>
    </bean>    -->
```

```
   <!-- <bean id="acc1" class="com.mainapp.Account"
></bean> -->
   <!-- <bean id="std1" class="com.mainapp.Student"
></bean> -->

   <!--  <bean id="employee" class="com.mainapp.Employee"
>
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="mapOfBooks" ref="myMap" >
</property>
    </bean>


     STANDALONE Map
     <util:map id="myMap" map-class="java.util.TreeMap"  >


       <entry key="book1010" value-ref="acc1"></entry>
       <entry key-ref="std1" value-ref="acc1"></entry>

       <entry key="book1010"
value="JAVABYCODEHUNT"></entry>
       <entry key="book1011"
value="SPRINGBYCODEHUNT"></entry>
       <entry key="book1012"
value="JAVAEEBYCODEHUNT"></entry>
    </util:map> -->


    <bean id="employee" class="com.mainapp.Employee" >
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="mapOfBooks" ref="myMap" >
</property>
    </bean>


    <bean id="myMap" class="java.util.TreeMap" >
```

```
        <constructor-arg>
            <map>
                <entry key="book1010"
value="JAVABYCODEHUNT"></entry>
                <entry key="book1011"
value="SPRINGBYCODEHUNT"></entry>
                <entry key="book1012"
value="JAVAEEBYCODEHUNT"></entry>
            </map>
        </constructor-arg>

    </bean>

</beans>
```

==========

==========

==========

Day-9

✅ Dependency Injection using XML Approach

✅ Properties Injection Through Setter Injection

- In Spring, java.util.Properties can be injected into a bean to represent a set of key-value pairs (like a map of strings). You use the <props> tag inside a <property> or <constructor-arg> element to inject it. Each <prop> tag represents a key-value pair. This is commonly used for configurations like error messages, labels, or database settings. Spring automatically creates a java.util.Properties object and populates it with the defined entries.

```
<bean id="config"
class="com.codehunt.Config">
    <property name="appProperties">
        <props>
            <prop
key="url">jdbc:mysql://localhost:3306/myd
b</prop>
```

```
            <prop
key="username">root</prop>
            <prop
key="password">admin</prop>
        </props>
    </property>
</bean>
```

- This injects a java.util.Properties object into the appProperties property using setter injection.
- Each <prop> tag represents a key-value pair (both must be Strings).
- This is commonly used for configuration properties like DB settings, labels, etc.
- Keys and values must be string literals — no <ref> support inside <props>.

✅ Reusable Properties with <util:properties> (Modular & Reusable Configuration)

```
<util:properties id="dbProps">
    <prop
key="driver">com.mysql.cj.jdbc.Driver</pr
op>
    <prop
key="url">jdbc:mysql://localhost:3306/myd
b</prop>
    <prop key="username">root</prop>
    <prop key="password">admin</prop>
</util:properties>

<bean id="dbConfig"
class="com.codehunt.DBConfig">
    <property name="properties"
ref="dbProps"/>
</bean>
```

- **This allows you to define a reusable Properties bean that can be injected into multiple beans.**
- **Very useful for shared configurations or app settings.**
- **Values must be string literals only.**

- Internally creates an instance of java.util.Properties.

✅ Properties Injection Using Separate <bean> Tag – Not Recommended

- In the case of java.util.Properties, using a separate <bean> tag with <constructor-arg> does not work as expected.
- Even if you define <props> inside <constructor-arg>, Spring may create the Properties object but won't populate it properly, and it will appear as an empty {} during injection.

✅ Properties Injection Through Constructor Injection

- In Spring, you can inject a Properties into a bean using constructor injection by passing the <props> element inside a <constructor-arg> tag.

Autowiring through XML

- Autowiring in Spring allows the container to automatically resolve dependencies between beans without explicitly specifying <property> or <constructor-arg> in the bean configuration.

✅ Types of Autowiring in XML

Type	Description
no	Default. No autowiring. You must define dependencies manually.

- byName ✅ Injects dependency by matching property name with bean id.
- byType ✅ Injects by matching property type with a bean of the same type.
- constructor ✅ Injects using constructor by matching parameter types.

- In Spring, the autowire="default" setting means that the bean will follow the default autowiring strategy defined at the container level <beans>. This is configured using the <beans default-autowire="..."> attribute. If a global default is set (e.g., byType), all beans with autowire="default" will follow that strategy.

✅ What is primary="true" in Spring?

- In Spring, when multiple beans of the same type exist, and autowiring is done by type or constructor (e.g., autowire="byType"), Spring gets

confused because it doesn’t know which bean to inject.

- To resolve this, we can mark one bean as the default choice using:

```
<bean id="mainService"
class="com.codehunt.ServiceImpl"
primary="true"/>
```

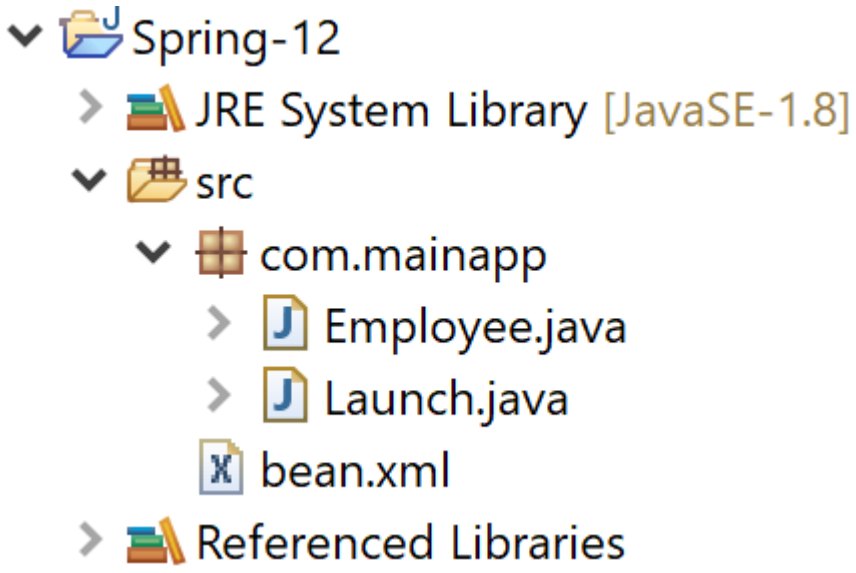
- Now, Spring will prefer this bean automatically for injection if multiple candidates are present.

How does autowire="constructor" work?

- When autowire="constructor" is used, Spring:
- Looks for the constructor with the most parameters it can satisfy (by matching bean types).
- Tries to match beans from the container by type to the constructor arguments.
- If multiple beans of the same type are available, it fails unless you use: primary="true" to prefer one

- **Autowire by constructor is like autowire by type, but instead of using setter methods, it uses the constructor to inject the dependencies.**

==========Program: Properties injection==========




```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;
public class Launch {

     public static void main(String[] args) {

          ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
```

```
          Employee employee=  (Employee)
ac.getBean("employee");
          System.out.println(employee);
          ac.close();

     }
}
```

```
package com.mainapp;
import java.util.Properties;
public class Employee {

     private int eid;
     private String ename;
     private Properties properties;

     public Employee() {
          System.out.println("EMP BEAN INSTANTIATION");
     }

     public void setEid(int eid) {
          this.eid = eid;
     }

     public void setEname(String ename) {
          this.ename = ename;
     }

     public int getEid() {
          return eid;
     }

     public String getEname() {
          return ename;
     }

     public Properties getProperties() {
          return properties;
     }

     public void setProperties(Properties properties) {
```

```
        this.properties = properties;
    }

    @Override
    public String toString() {
        return "Employee [eid=" + eid + ", ename=" +
ename + ", properties=" + properties + "]";
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.x
sd
       http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd
"> <!-- bean definitions here -->


 <!--   <bean id="employee" class="com.mainapp.Employee" >
       <property name="eid"  value="123412"></property>
       <property name="ename"  value="raju"></property>
       <property name="properties">
            <props>
               <prop
key="url">jdbc:mysql://localhost:8080</prop>
               <prop key="user">root</prop>
               <prop key="password">admin</prop>
            </props>
       </property>
    </bean>    -->
```

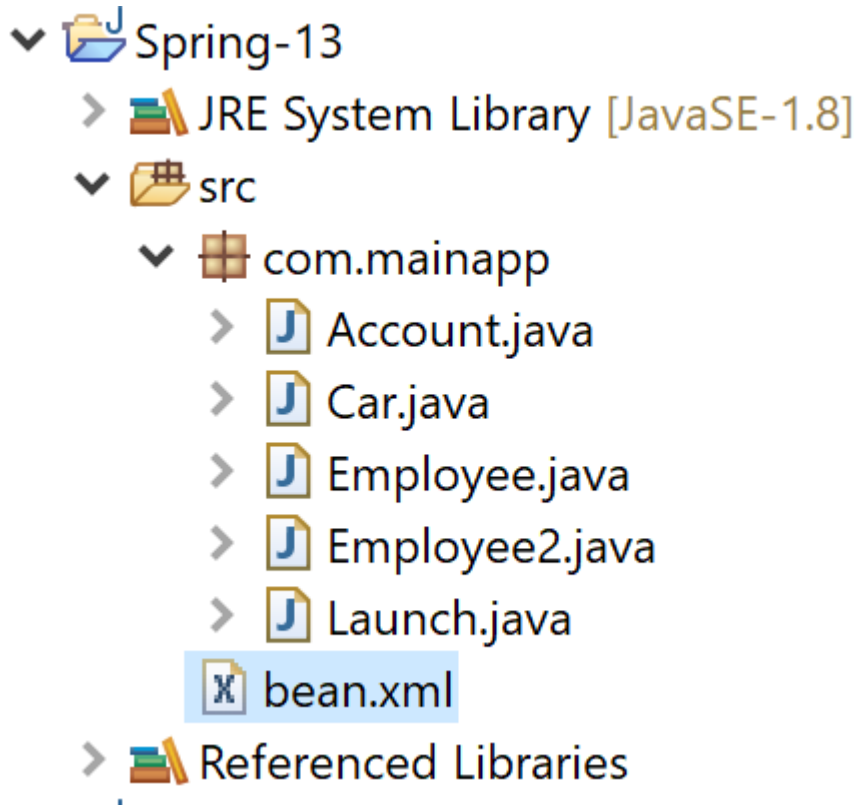
```
     <bean id="employee" class="com.mainapp.Employee" >
        <property name="eid"  value="123412"></property>
        <property name="ename"  value="raju"></property>
        <property name="properties" ref="myProperties"
></property>
     </bean>

     <!-- STANDALONE Properties -->
     <util:properties id="myProperties">
         <prop
key="url">jdbc:mysql://localhost:8080</prop>
         <prop key="user">root</prop>
         <prop key="password">admin</prop>
     </util:properties>
</beans>
```

==========

==========

==========

==========Program : Autowiring in XML==========




```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("bean.xml");
		Employee employee=  (Employee)
ac.getBean("employee");
		System.out.println(employee);

		Employee2 employee2=  (Employee2)
ac.getBean("employee2");
		System.out.println(employee2);
		ac.close();

	}
}
```

```java
package com.mainapp;
import java.util.List;
public class Employee {

	private int eid;
	private Account account;
	private Car car;
	private List<String> listOfBooks;

	public Employee() {
		// TODO Auto-generated constructor stub
	}

	public Employee(Account account, Car car,
List<String> listOfBooks) {
		super();
		this.account = account;
		this.car = car;
		this.listOfBooks = listOfBooks;
	}

	public int getEid() {
		return eid;
	}

	public void setEid(int eid) {
		this.eid = eid;
	}

	public Account getAccount() {
		return account;
	}

	public void setAccount(Account account) {
		this.account = account;
	}

	public Car getCar() {
		return car;
	}

	public void setCar(Car car) {
```

```
		this.car = car;
	}

	public List<String> getListOfBooks() {
		return listOfBooks;
	}

	public void setListOfBooks(List<String> listOfBooks)
{
		this.listOfBooks = listOfBooks;
	}

	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", account=" +
account + ", car=" + car + ", listOfBooks=" + listOfBooks
+ "]";
	}
}
package com.mainapp;
import java.util.List;
public class Employee2 {

	private int eid;
	private Account account;
	private Car car;
	private List<String> listOfBooks;

	public Employee2() {
		// TODO Auto-generated constructor stub
	}

	public Employee2(Account account, Car car,
List<String> listOfBooks) {
		super();
		this.account = account;
		this.car = car;
		this.listOfBooks = listOfBooks;
	}

	public int getEid() {
		return eid;
```

```
	}

	public void setEid(int eid) {
		this.eid = eid;
	}

	public Account getAccount() {
		return account;
	}

	public void setAccount(Account account) {
		this.account = account;
	}

	public Car getCar() {
		return car;
	}

	public void setCar(Car car) {
		this.car = car;
	}

	public List<String> getListOfBooks() {
		return listOfBooks;
	}

	public void setListOfBooks(List<String> listOfBooks)
{
		this.listOfBooks = listOfBooks;
	}

	@Override
	public String toString() {
		return "Employee2 [eid=" + eid + ", account=" +
account + ", car=" + car + ", listOfBooks=" + listOfBooks
+ "]";
	}
}
package com.mainapp;

public class Car {
```

```
	private int cn;
	private String carName;

	public Car() {
		// TODO Auto-generated constructor stub
	}

	public Car(int cn, String carName) {
		super();
		this.cn = cn;
		this.carName = carName;
	}

	public int getCn() {
		return cn;
	}

	public void setCn(int cn) {
		this.cn = cn;
	}

	public String getCarName() {
		return carName;
	}

	public void setCarName(String carName) {
		this.carName = carName;
	}

	@Override
	public String toString() {
		return "Car [cn=" + cn + ", carName=" + carName +
"]";
	}
}
package com.mainapp;
public class Account {

	private int an;
	private String bankName;

	public Account() {
```

```
	}

	public Account(int an, String bankName) {
		super();
		this.an = an;
		this.bankName = bankName;
	}

	public int getAn() {
		return an;
	}

	public void setAn(int an) {
		this.an = an;
	}

	public String getBankName() {
		return bankName;
	}

	public void setBankName(String bankName) {
		this.bankName = bankName;
	}

	@Override
	public String toString() {
		return "Account [an=" + an + ", bankName=" +
bankName + "]";
	}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    default-autowire="byName"

    xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:util="http://www.springframework.org/schema/util"
     xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.x
sd
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd
"> <!-- bean definitions here -->



        <!-- MANULA WIRING -->
       <!--  <bean id="employee"
class="com.mainapp.Employee" >
              <property name="eid"
value="123412"></property>
              <property name="account"  ref="account"
></property>
              <property name="car"   ref="car"></property>
              <property name="listOfBooks"
ref="listOfBooks" ></property>
        </bean>

        <bean id="account" class="com.mainapp.Account" >
            <property name="an" value="121212" >
</property>
            <property name="bankName" value="xyzBank" >
</property>
         </bean>

         <bean id="car" class="com.mainapp.Car" >
            <property name="cn" value="9999" > </property>
            <property name="carName" value="xyzCar" >
</property>
         </bean>

         <util:list id="listOfBooks">
           <value>JavaByCodehunt</value>
           <value>SpringByCodehunt</value>
         </util:list> -->

        <!--
=============================================================
===== -->
```

```
        <!-- Autowiring byName -->
        <!--  <bean id="employee"
class="com.mainapp.Employee" autowire="byName" >
              <property name="eid"
value="123412"></property>
        </bean>

         <bean id="account" class="com.mainapp.Account" >
            <property name="an" value="121212" >
</property>
            <property name="bankName" value="xyzBank" >
</property>
         </bean>

         <bean id="car" class="com.mainapp.Car" >
            <property name="cn" value="9999" > </property>
            <property name="carName" value="xyzCar" >
</property>
         </bean>

         <util:list id="listOfBooks">
           <value>JavaByCodehunt</value>
           <value>SpringByCodehunt</value>
         </util:list> -->


         <!--
========================================================
===== -->


        <!-- Autowiring byType -->

       <!--  <bean id="employee"
class="com.mainapp.Employee" autowire="byType" >
              <property name="eid"
value="123412"></property>
        </bean>
```

```
         <bean id="account1" class="com.mainapp.Account"
primary="true" >
            <property name="an" value="1212121"  >
</property>
            <property name="bankName" value="xyzBank1" >
</property>
         </bean>

          <bean id="account2" class="com.mainapp.Account"
>
            <property name="an" value="1212122" >
</property>
            <property name="bankName" value="xyzBank2" >
</property>
         </bean>

         <bean id="car1" class="com.mainapp.Car" >
            <property name="cn" value="9999" > </property>
            <property name="carName" value="xyzCar" >
</property>
         </bean>

         <util:list id="listOfBooks1">
           <value>JavaByCodehunt</value>
           <value>SpringByCodehunt</value>
         </util:list> -->


         <!--
========================================================
===== -->


        <!-- Autowiring constructor -->

        <!--  <bean id="employee"
class="com.mainapp.Employee" autowire="constructor" >
              <property name="eid"
value="123412"></property>
        </bean>
```

```
         <bean id="account1" class="com.mainapp.Account"
primary="true" >
            <property name="an" value="12121211" >
</property>
            <property name="bankName" value="xyzBank1" >
</property>
         </bean>

         <bean id="account2" class="com.mainapp.Account"
>
            <property name="an" value="12121222" >
</property>
            <property name="bankName" value="xyzBank2" >
</property>
         </bean>

         <bean id="car1" class="com.mainapp.Car" >
            <property name="cn" value="9999" > </property>
            <property name="carName" value="xyzCar" >
</property>
         </bean>

         <util:list id="listOfBooks1">
           <value>JavaByCodehunt</value>
           <value>SpringByCodehunt</value>
         </util:list>
         -->


        <!--
========================================================
===== -->


        <!-- Autowiring default & No -->

        <bean id="employee" class="com.mainapp.Employee"
autowire="default">
              <property name="eid"
value="1234111"></property>
        </bean>
```

```
        <bean id="employee2" class="com.mainapp.Employee2"
autowire="byType" >
              <property name="eid"
value="1234222"></property>
        </bean>

         <bean id="account" class="com.mainapp.Account"
primary="true" >
            <property name="an" value="12121211" >
</property>
            <property name="bankName" value="xyzBank1" >
</property>
         </bean>

         <bean id="car" class="com.mainapp.Car" >
            <property name="cn" value="9999" > </property>
            <property name="carName" value="xyzCar" >
</property>
         </bean>

         <util:list id="listOfBooks">
           <value>JavaByCodehunt</value>
           <value>SpringByCodehunt</value>
         </util:list>

</beans>
```

==========

==========

==========

Day-10

✅ Why Use XML Configuration in Spring?

So far, we've explored the XML-based configuration approach. It is still relevant and useful in many scenarios:

- **Separation of Concerns: XML allows you to keep configuration separate from the source code, which helps in managing dependencies centrally and clearly.**
- **Legacy Application Support: If you're working with existing legacy systems that already use XML, it's often better to stick with XML for consistency and easier maintenance.**

✅ Then Why Learn Java-based Configuration (Annotations)?

Despite the benefits of XML, Java-based configuration is widely preferred in modern Spring applications for several reasons:

- **Simplicity and Readability: Annotations like @Component, @Configuration, and @Bean make the code cleaner and easier to understand.**
- **Less Boilerplate: You write less code compared to XML, reducing chances of errors and making development faster.**
- **Type-Safety: Since configuration is written in Java, it benefits from compile-time checking and IDE support like auto-completion and refactoring.**
- **Easier to Maintain: Changes can be made directly in the source without having to switch between XML files and Java code.**
- **Widely Adopted: Most modern Spring Boot applications use annotation-based configuration as a standard practice.**

✅ Conclusion:

- **While XML is still useful in some cases (especially for legacy systems or externalizing**

configurations), annotation-based Java configuration is the preferred choice in modern development due to its simplicity

✅ Best Practices in Spring Configuration

- **Prefer a Hybrid Approach (XML + Annotations)**
  → Combine the strengths of both: use annotations for bean management and XML for external/environment-specific settings.
- **Use Annotations for Bean Definitions and Dependency Injection**
  → Annotations like @Component, @Autowired, @Service, etc., make your code cleaner, more readable, and easier to maintain.
- **Use XML for Environment-Specific Configuration (in traditional Spring)**
  → You can externalize settings like database configurations (URL, username, password, driver class, etc.) in XML so they can be changed easily without modifying the source code.

✅ What We Will Learn with Annotation-Based Configuration in Spring

With annotations, we will explore how to manage and control beans in a clean and modern way:

- **Configuration**
  - **Using @Configuration and @Bean to define and manage beans in Java.**
- **Object Creation (Simple & Custom Logic)**
  - **Creating objects with default constructors or custom logic using @Bean methods.**
- **Lazy vs Eager Initialization**
  - **Understanding how beans are created at startup or on demand using @Lazy.**
- **Bean Scope**
  - **Managing bean lifecycles with scopes like singleton, prototype, request, session using @Scope.**
- **Bean Life Cycle**

- Hooking into initialization and destruction using @PostConstruct, @PreDestroy, and lifecycle interfaces.

- **BeanPostProcessor**
  - Customizing bean creation process with BeanPostProcessor to apply logic before/after initialization.
- **Dependency Injection & SpEL (Spring Expression Language)**
  - Using @Autowired, @Qualifier, @Value for injecting dependencies and leveraging SpEL for dynamic value injection.

✅

AnnotationConfigApplicationContext – Key Points

- **It's a Spring Container**
  → A specialized ApplicationContext implementation designed for Java-based configuration using annotations.

- **Introduced in Spring 3.0**

  → **Provides support for @Configuration, @ComponentScan, @Bean, @Component, etc.**
- **No Need for XML Files**

→ **Enables you to create and manage beans entirely using Java code.**

✅ Use @Bean When:

- **You Don't Have Access to Source Code**
- **Example: Creating a bean from a third-party class or library that you cannot annotate with @Component.**

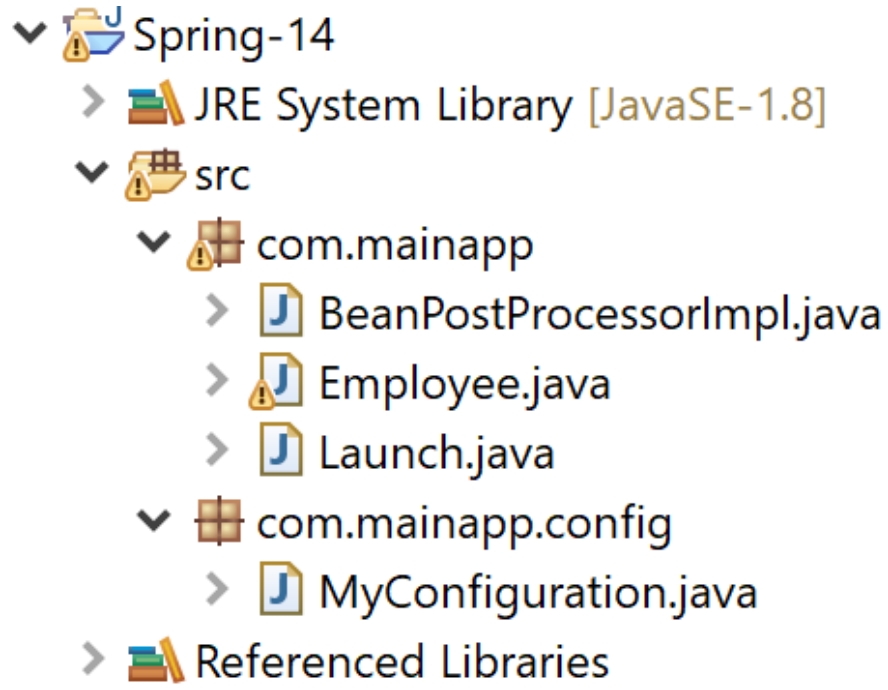
```
@Bean
public DataSource dataSource() {
    return new BasicDataSource();
}
```

- **You Want Full Control Over Object Creation**

- **You can add custom logic, constructor parameters, conditionals, or logging before returning the object.**

```
@Bean
public Employee employee() {
    System.out.println("Custom object creation logic here");
    return new Employee();
}
```

==========Program : Annotation Approach==========




```
package com.mainapp;
import
org.springframework.context.annotation.AnnotationConfigApp
licationContext;
import com.mainapp.config.MyConfiguration;
public class Launch {

	public static void main(String[] args) {

		AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(MyConfiguration.class);
		//Employee bean = (Employee) ac.getBean("emp");
		Employee bean1 = ac.getBean(Employee.class);
		System.out.println(bean1);
		bean1.test();

		Employee bean2 = ac.getBean(Employee.class);
		System.out.println(bean2);
		bean2.test();

		ac.close();
	}
}
package com.mainapp;
```

```java
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

//@Component("emp")
//@Component
//@Lazy
//@Scope("prototype")
public class Employee {

	public Employee() {
		System.out.println("BEAN INSTANTIATION");
	}

	//@PostConstruct
	public void init() {
		System.out.println("BEAN INIT");
	}

	//@PreDestroy
	public void destroy() {
		System.out.println("BEAN DESTORY");
	}

	public void test() {
		System.out.println("BEAN TESTED");
	}
}
package com.mainapp;
import org.springframework.beans.BeansException;
import
org.springframework.beans.factory.config.BeanPostProcessor
;
import org.springframework.stereotype.Component;

@Component
public class BeanPostProcessorImpl implements
BeanPostProcessor{

	@Override
```

```
	public Object postProcessBeforeInitialization(Object
bean, String beanName) throws BeansException {


System.out.println("postProcessBeforeInitialization");
		return bean;
	}

	@Override
	public Object postProcessAfterInitialization(Object
bean, String beanName) throws BeansException {

System.out.println("postProcessAfterInitialization");
		return bean;
	}
}
package com.mainapp.config;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.ComponentScan;
import
org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Scope;

import com.mainapp.Employee;

@Configuration
@ComponentScan(basePackages = {"com.mainapp"})
public class MyConfiguration {

	@Bean(initMethod = "init" , destroyMethod =
"destroy")
	@Lazy
	@Scope("prototype")
	public Employee getEmployee() {
		System.out.println("CUSTOM LOGIC");
		return new Employee();
	}
}
```

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

Day-11

✅ Dependency Injection (Java Based Configuration)

- In Spring, Dependency Injection (DI) is the process of providing the dependencies (objects or values) a class needs, instead of the class creating them on its own. DI promotes loose coupling and makes the code easier to manage, test, and maintain.
- There are three main types of injection:
    1. Setter Injection
    2. Constructor Injection
    3. Field Injection

NOTE:

- ***Before starting Dependency Injection, remember that in autowiring, we inject only beans (objects) — not primitive values or simple Strings. To inject such values, Spring provides a separate annotation called @Value, and for evaluating dynamic expressions, it uses a feature called SpEL (Spring Expression Language).***

Setter Injection

- **Setter Injection is when Spring injects a dependency through a public setter method.**

How it Works Internally:

- **Spring creates the object using the no-arg constructor.**
- **Then, it calls the setter method and injects the required dependency.**

```
@Component
public class Student {
    private Address address;

    @Autowired  // Autowiring by type
    public void setAddress(Address address)
{
        this.address = address;
    }
    public void show() {
        System.out.println("City: " +
address.getCity());
    }
}
```

Constructor Injection

- **Constructor Injection is when Spring injects dependencies via constructor parameters.**

How it Works Internally:

- **Spring identifies the constructor with parameters.**
- **It looks for matching beans in the container by type and injects them while creating the object.**

```
@Component
public class Teacher {
    private Address address;

    @Autowired  // Optional (since Spring 4.3+ with single constructor)
    public Teacher(Address address) {
        this.address = address;
    }
    public void display() {
        System.out.println("City: " + address.getCity());
    }
```

```
}
```

Field Injection

- **Field Injection is when Spring directly injects the dependency into the field.**

How it Works Internally:

- **Spring uses reflection to set the value of the private field.**
- **No setter or constructor needed.**

```
@Component
public class Course {
    @Autowired
    private Address address;

    public void print() {
        System.out.println("City: " +
address.getCity());
    }
```

```
}
```

@Qualifier Annotation

- In setter, constructor, and field injection, when we use @Autowired, Spring first looks for a matching bean by type (like Account). If only one bean of that type is found, it injects it. But if there are multiple beans of the same type, Spring then checks the name of the field or parameter (like account) to find the right one. So, autowiring always starts by type and then uses the name if needed. This rule is the same for all three types of injection.
- To make Spring autowire a bean specifically by name, you can use the @Qualifier annotation along with @Autowired. When there are multiple beans of the same type, @Qualifier("beanName") tells Spring exactly which bean to inject by matching the name. This works with setter, constructor, and field injection.

@Primary Annotation

- In Spring, the @Primary annotation is used when there are multiple beans of the same type, and you want one of them to be the default choice for autowiring. When Spring finds more than one matching bean during dependency injection, it uses the bean marked with @Primary automatically, without requiring a @Qualifier. This helps avoid ambiguity errors. You can still use @Qualifier to explicitly choose a different bean if needed. @Primary is useful when one bean should be preferred most of the time, but others might be used occasionally.

Collection Injection in Annotation-Based Configuration (Spring)

- **Collection Injection in Spring is the process of injecting collection-type objects such as List, Set, Map, Queue, or Stack into a class. This is especially useful when a class needs to work with multiple values or a group of related objects instead of a single dependency.**

✅ How to Inject Collections Using Annotations?

In Spring's Java-based configuration, you can inject collections using:

- **@Configuration and @Bean to define the collection**
- **@Autowired to inject the collection**
- **(Optional) @Qualifier or @Primary if multiple beans of the same type exist**

You can inject the collection using any method — setter injection, constructor injection, or field injection, depending on your design preference.

✅ Step-by-Step Example: Injecting a List

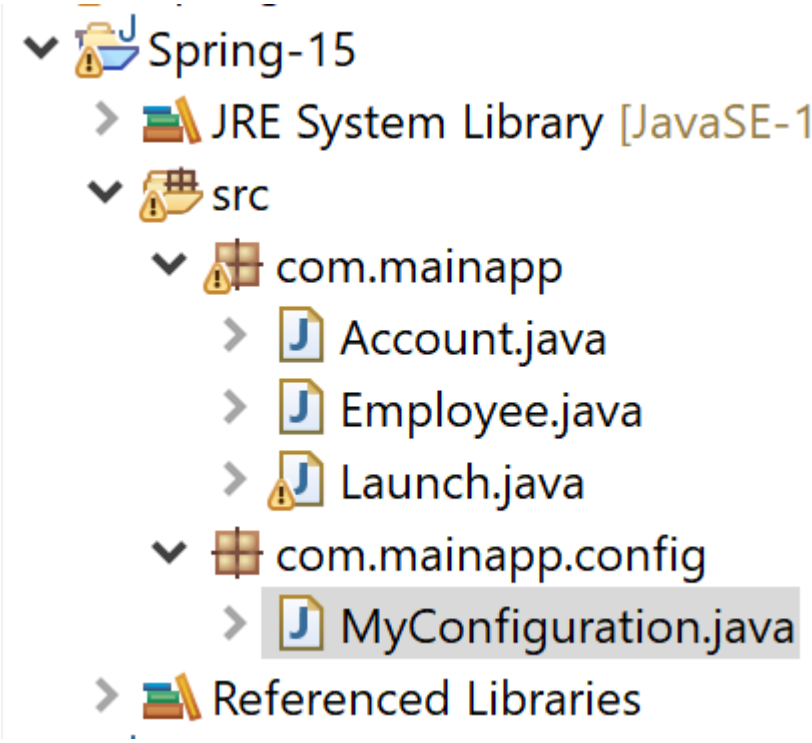
- **Define Bean in @Configuration**

```
@Configuration
public class AppConfig {
    @Bean
    public List<String> subjects() {
        return Arrays.asList("Java",
"Spring", "Hibernate");
    }
}
```

- **Inject Using @Autowired**

```
@Component
public class Student {
    @Autowired
    private List<String> subjects;

    public void showSubjects() {
        System.out.println(subjects);
    }
}
```

==========Program: @Autowire @Qualifier @Primary==========




```
package com.mainapp;
import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;
import com.mainapp.config.MyConfiguration;
public class Launch {

	public static void main(String[] args) {

		AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(MyConfiguration.class);
		Employee bean = ac.getBean(Employee.class);
		System.out.println(bean);
	}
```

```
}
```

```
package com.mainapp;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class Employee {

	@Autowired
	private Account account;

	@Autowired
	@Qualifier("list1")
	private List<String> listOfBooks;

	public Employee() {
		System.out.println("EMP BEAN INSTANTIATION: ZPC");
	}

	//@Autowired
	public Employee(Account account, List<String> listOfBooks) {
		super();
		System.out.println("EMP BEAN INSTANTIATION: PC");
		this.account = account;
		this.listOfBooks = listOfBooks;
	}


	public Account getAccount() {
		return account;
	}


	public List<String> getListOfBooks() {
		return listOfBooks;
	}

	//@Autowired
	public void setListOfBooks(List<String> listOfBooks) {
		System.out.println("EMP SETTER CALL: LIST");
		this.listOfBooks = listOfBooks;
	}

	//@Autowired
```

```
	public void setAccount(Account account) {
		System.out.println("EMP SETTER CALL: ACCOUNT");
		this.account = account;
	}

	@Override
	public String toString() {
		return "Employee [account=" + account + ", listOfBooks="
+ listOfBooks + "]";
	}
}
```

```
package com.mainapp;
import org.springframework.stereotype.Component;
@Component
public class Account {

	private int an=10;
	private String bankName="xyzbank";

	public Account() {
		System.out.println("ACC BEAN INSTANTIATION: ZPC");
	}

	public Account(int an, String bankName) {
		super();
		System.out.println("ACC BEAN INSTANTIATION: PC");
		this.an = an;
		this.bankName = bankName;
	}

	public int getAn() {
		return an;
	}

	public void setAn(int an) {
		this.an = an;
	}

	public String getBankName() {
		return bankName;
	}

	public void setBankName(String bankName) {
		this.bankName = bankName;
	}
```

```
	@Override
	public String toString() {
		return "Account [an=" + an + ", bankName=" + bankName +
"]";
	}
}
package com.mainapp.config;
import java.util.Arrays;
import java.util.List;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
@Configuration
@ComponentScan(basePackages = {"com.mainapp"})
public class MyConfiguration {

	@Bean("list1")
	public List<String> getList1(){
	   return  Arrays.asList("b1","b2","b3");
	}

	@Bean("list2")
	@Primary
	public List<String> getList2(){
	   return  Arrays.asList("b11","b22","b33");
	}

	@Bean("list3")
	public List<String> getList3(){
	   return  Arrays.asList("b111","b222","b333");
	}

}
```

==========

==========

==========

@Value Annotation

- **@Value is a Spring annotation used to inject literal values, property file values, or SpEL (Spring Expression Language) expressions into a Spring bean.**
- **It is a form of dependency injection, but instead of injecting beans (objects), it injects primitive values, Strings, or evaluated expressions.**
- **Spring can inject these values using:**
    1. **Field Injection**
    2. **Setter Injection**
    3. **Constructor Injection (parameters)**
- **So, @Value supports all standard injection types used in Spring.**

✅ Inject Literal Values (Primitive or String)

```
@Value("Java Full Stack")
private String courseName;


@Value("6999")
```

```
private int courseFee;


@Value("true")
private boolean isActive;
```

✔ These are hardcoded values.

✅ Inject Values from .properties File

File: app.properties

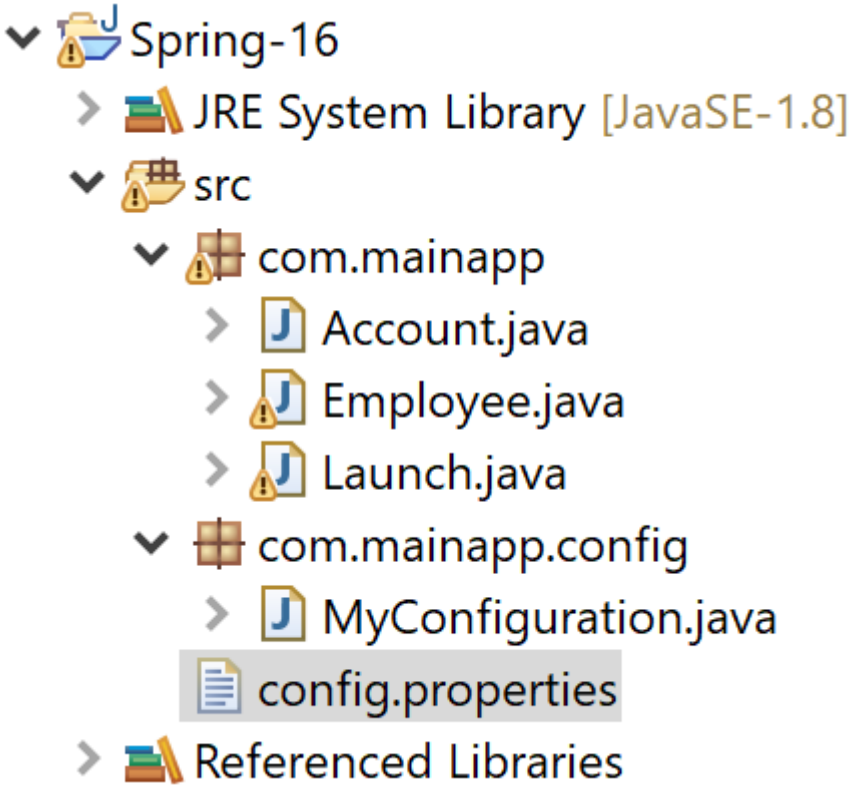
```
course.name=Java Full Stack
course.fee=6999
course.active=true
```

```
@Value("${course.name}")
private String courseName;
```

To load a .properties file from the classpath in Spring, place it under the src directory (which is the classpath). Then use @PropertySource("classpath:filename.properties") in a @Configuration class to make its values accessible.

==========

==Program: @Value==========




```
package com.mainapp;
```

```
import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;
import com.mainapp.config.MyConfiguration;
public class Launch {

	public static void main(String[] args) {

		AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(MyConfiguration.class);
		Employee bean = ac.getBean(Employee.class);
		System.out.println(bean);
	}
}

package com.mainapp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@Component
public class Employee {

	//@Value("1000")
	private int eid;
	//@Value("raju")
	private String ename;

	//@Value("${db.url}")
	private String url;

	public Employee() {
		System.out.println("EMP BEAN INST: ZPC");
	}

	@Autowired
	public Employee(@Value("1000")int eid, @Value("raju")String
ename, @Value("${db.url}")String url) {
		super();
		System.out.println("EMP BEAN INST: PC");
		this.eid = eid;
		this.ename = ename;
		this.url = url;
	}

	public int getEid() {
		return eid;
	}

	//@Value("1000")
```

```
	public void setEid(int eid) {
		System.out.println("EMP BEAN SETTER: eid");
		this.eid = eid;
	}

	public String getEname() {
		return ename;
	}

	// @Value("raju")
	public void setEname(String ename) {
		System.out.println("EMP BEAN SETTER: ename");
		this.ename = ename;
	}

	public String getUrl() {
		return url;
	}

	//@Value("${db.url}")
	public void setUrl(String url) {
		System.out.println("EMP BEAN SETTER: url");
		this.url = url;
	}

	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" + ename + ",
url=" + url + "]";
	}

}
```

```
package com.mainapp;
import org.springframework.stereotype.Component;

@Component
public class Account {

	private int an=10;
	private String bankName="xyzbank";

	public Account() {
		System.out.println("ACC BEAN INSTANTIATION: ZPC");
	}

	public Account(int an, String bankName) {
```

```
            super();
            System.out.println("ACC BEAN INSTANTIATION: PC");
            this.an = an;
            this.bankName = bankName;
      }

      public int getAn() {
            return an;
      }

      public void setAn(int an) {
            this.an = an;
      }

      public String getBankName() {
            return bankName;
      }

      public void setBankName(String bankName) {
            this.bankName = bankName;
      }

      @Override
      public String toString() {
            return "Account [an=" + an + ", bankName=" + bankName +
"]";
      }
}
```

```
package com.mainapp.config;
import java.util.Arrays;
import java.util.List;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.PropertySource;
@Configuration
@ComponentScan(basePackages = {"com.mainapp"})
@PropertySource("classpath:config.properties")
public class MyConfiguration {

      @Bean("list1")
      public List<String> getList1(){
         return  Arrays.asList("b1","b2","b3");
      }
```

```
    @Bean("list2")
    @Primary
    public List<String> getList2(){
       return  Arrays.asList("b11","b22","b33");
    }

    @Bean("list3")
    public List<String> getList3(){
       return  Arrays.asList("b111","b222","b333");
    }

}
```

config.properties

```
1 db.url=jdbc:mysql://localhost:8080
```

==========

==========

==========

What is SpEL?

SpEL (Spring Expression Language) is a powerful expression language in Spring used to evaluate:

- **Mathematical expressions**
- **Logical conditions**

- Access bean properties
- Call methods
- Work with collections and arrays
- Use ternary operators, etc.

✅ SpEL Examples (One by One)

1. Literal Values

```
@Value("#{42}")
private int num; // Injects 42
```

2. Mathematical Expressions

```
@Value("#{10 + 5 * 2}")
private int result; // Injects 20
```

3. Calling Static Methods

```
@Value("#{T(java.lang.Math).random()}")
```

```
private double rand; // Injects a random
double between 0.0 to 1.0
```

4. Calling Instance Methods (from other beans)

- **Assuming a bean employee with method getName():**

```
@Value("#{employee.getName()}")
private String empName; // Injects the
name from employee bean
```

5. Ternary Operator

```
@Value("#{test.checkSalary() > 5000 ?
'High' : 'Low'}")
private String level; // Injects "High"
or "Low" based on `salary`
```

6. Accessing Properties File Using SpEL and Environment

```
@Value("#{environment['app.name']}")
private String appName;  // Fetches value from application.properties
```

To load a .properties file from the classpath in Spring, place it under the src directory (which is the classpath). Then use @PropertySource("classpath:filename.properties") in a @Configuration class to make its values accessible.

✅ Collections Supported in SpEL (Spring Expression Language):

SpEL supports all standard Java collection types:

✅ List

```
@Value("#{{'Java', 'Spring', 'Hibernate'}}")
private List<String> topic;
```

✅ Set

```
@Value("#{ {'A', 'B', 'C'} }")  // Interpreted as a Set if uniqueness matters

private Set<String> letters;
```

✅ Map

```
@Value("#{ {'key1':'Java', 'key2':'Spring'} }")

private Map<String, String> techMap;
```

✅ Array

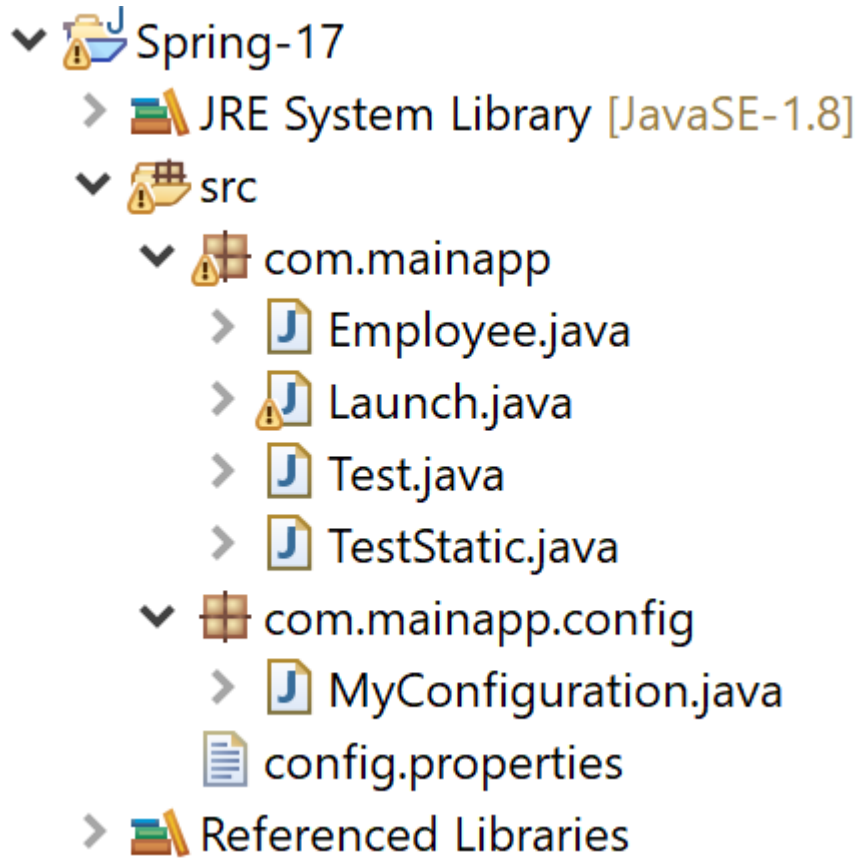
```
@Value("#{ new int[]{10, 20, 30} }")

private int[] numbers;  // Injects entire array
```

==========

Program:

SPEL==========

==========




```
package com.mainapp;
import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;
import com.mainapp.config.MyConfiguration;
public class Launch {

	public static void main(String[] args) {

		AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(MyConfiguration.class);
		Employee bean = ac.getBean(Employee.class);
		int[] array = bean.getArray();
		for(int k : array)
			System.out.println(k);
		System.out.println(bean);
	}
}
```

```
package com.mainapp;
import java.util.List;
import java.util.Map;
```

```java
import java.util.Set;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@Component
public class Employee {

	@Value("#{42+82}")
	private int eid;

	//@Value("#{'raju'}")
	//@Value("#{test.doTest()}")
	//@Value("#{T(com.mainapp.TestStatic).doTestStatic() }")

	@Value("#{  test.checkSalary()>500 ? 'Valid' : 'invalid'   }")
	private String ename;

	@Value("#{       {'b1','b2','b3'}       }")
	private List<String> list;

	@Value("#{       {'b1','b2','b3'}       }")
	private Set<String> set;

	@Value("#{       {'key1':'b1','key2':'b2','key3':'b3'}
}")
	private Map<String,String> map;

	@Value("#{ new int[]{10,20,30} }")
	private int[] array;


	@Value("#{environment['db.url']}")
	private String url;

	public Employee() {
		System.out.println("EMP BEAN INST: ZPC");
	}

	public Employee(int eid, String ename, List<String> list,
Set<String> set, Map<String, String> map, int[] array,
			String url) {
		super();
		this.eid = eid;
		this.ename = ename;
		this.list = list;
		this.set = set;
		this.map = map;
		this.array = array;
		this.url = url;
	}
```

```java
public int getEid() {
	return eid;
}

public void setEid(int eid) {
	System.out.println("EMP BEAN SETTER: eid");
	this.eid = eid;
}

public String getEname() {
	return ename;
}

public void setEname(String ename) {
	System.out.println("EMP BEAN SETTER: ename");
	this.ename = ename;
}

public String getUrl() {
	return url;
}

public void setUrl(String url) {
	System.out.println("EMP BEAN SETTER: url");
	this.url = url;
}

public List<String> getList() {
	return list;
}

public void setList(List<String> list) {
	this.list = list;
}

public Set<String> getSet() {
	return set;
}

public void setSet(Set<String> set) {
	this.set = set;
}

public Map<String, String> getMap() {
	return map;
}
```

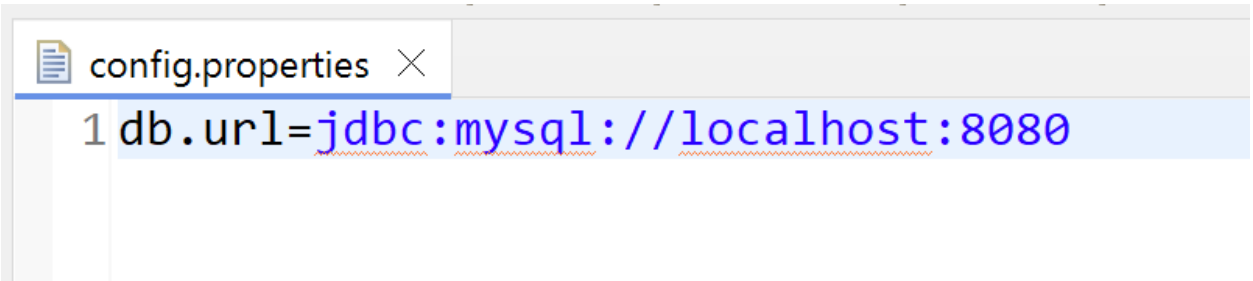
```
	public void setMap(Map<String, String> map) {
		this.map = map;
	}

	public int[] getArray() {
		return array;
	}

	public void setArray(int[] array) {
		this.array = array;
	}

	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" + ename + ",
list=" + list + ", set=" + set + ", map=" + map
				+ ", array=" + array + ", url=" + url + "]";
	}

}
```

```
package com.mainapp;
import org.springframework.stereotype.Component;

@Component("test")
public class Test {

	public String doTest() {
		return "test";
	}

	public int checkSalary() {
		return 100;
	}

}
```

```
package com.mainapp;

public class TestStatic {

	public static String doTestStatic() {
		return "doTestStatic";
	}
```

```
}
```

```
package com.mainapp.config;
import java.util.Arrays;
import java.util.List;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.PropertySource;
@Configuration
@ComponentScan(basePackages = {"com.mainapp"})
@PropertySource("classpath:config.properties")
public class MyConfiguration {

	@Bean("list1")
	public List<String> getList1(){
	   return  Arrays.asList("b1","b2","b3");
	}

	@Bean("list2")
	@Primary
	public List<String> getList2(){
	   return  Arrays.asList("b11","b22","b33");
	}

	@Bean("list3")
	public List<String> getList3(){
	   return  Arrays.asList("b111","b222","b333");
	}

}
```

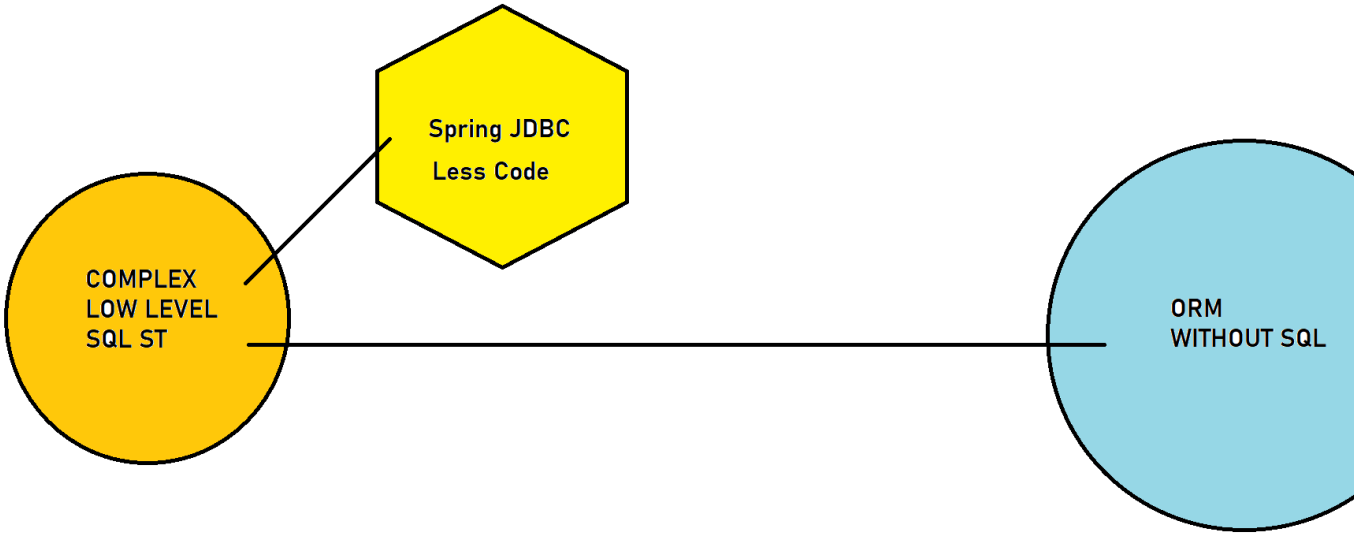


==========
==========
==========

Day-12

Spring JDBC

Spring JDBC is a part of the Spring Framework that simplifies JDBC (Java Database Connectivity) operations. JDBC is used in Java to connect and interact with databases, but traditional JDBC code is verbose and full of boilerplate like:



- Opening/closing database connections
- Handling exceptions
- Managing transactions
- Writing try-catch-finally blocks

What Spring JDBC does:

Spring JDBC removes much of this boilerplate and makes it easier and cleaner to perform database operations.

✅ Core Features:

- Simplifies JDBC code using JdbcTemplate
- Automatic resource management (no need to manually close connections)

- Better exception handling (converts checked SQLExceptions into runtime exceptions)
- Transaction management support
- Works well with Spring's dependency injection

JdbcTemplate – Core Class in Spring JDBC

JdbcTemplate is the central class in the Spring JDBC module. It simplifies interaction with relational databases by handling:

- **Database connections**
- **SQL execution**
- **Parameter binding**
- **Exception translation**
- **Resource cleanup**

You no longer need to write repetitive boilerplate code like opening/closing connections or handling SQLExceptions.

❓ Does JdbcTemplate provide a pool of objects?

- **No, JdbcTemplate itself does NOT provide a connection pool.**

CRUD OPERATIONS using Spring JDBC with JdbcTemplate

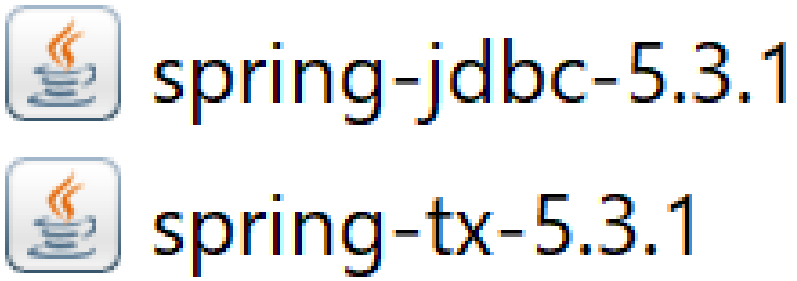
Spring JDBC provides the JdbcTemplate class to perform all CRUD operations (Create, Read, Update, Delete) on the database easily and efficiently.

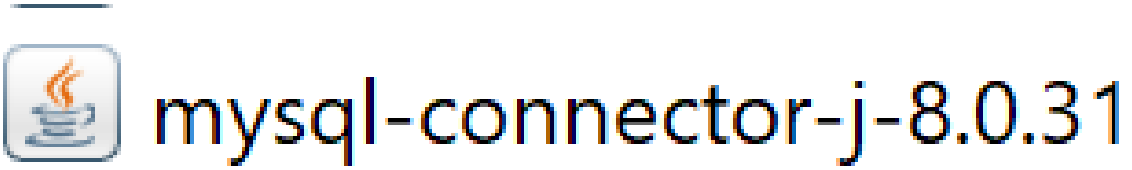
✅ Two Ways to Configure JdbcTemplate

1. XML Configuration

2. Annotation-Based Configuration (@Configuration, @Bean, @Autowired)

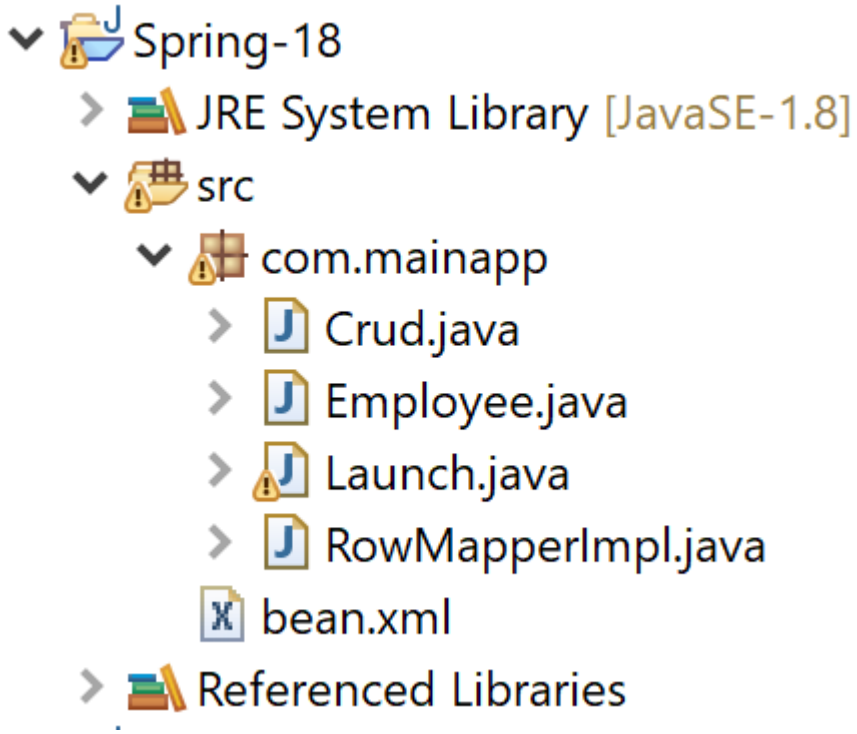
To work with Spring JDBC, you need three essential JARs: spring-jdbc , spring-tx and Connector.





- **Program1: XML Configuration**
- **Program2: Annotation Configuration**

==========Program: Spring JDBC (XML APPROACH)==========




```
package com.mainapp;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Launch {

	public static void main(String[] args) {

		ClassPathXmlApplicationContext ac=new
ClassPathXmlApplicationContext("bean.xml");
		Crud bean = (Crud) ac.getBean("crud");
		//bean.insert(16,"raju","add1",1000);
		//bean.update(16, "kaju");
		//bean.delete(16);
		//bean.readAll();
		//bean.readAllPojoBased();
		bean.readSingleData(14);


	}
}
package com.mainapp;
public class Employee {

	private int eid;
	private String ename;
	private String eaddress;
	private int esalary;
```

```
	public int getEid() {
		return eid;
	}
	public void setEid(int eid) {
		this.eid = eid;
	}
	public String getEname() {
		return ename;
	}
	public void setEname(String ename) {
		this.ename = ename;
	}
	public String getEaddress() {
		return eaddress;
	}
	public void setEaddress(String eaddress) {
		this.eaddress = eaddress;
	}
	public int getEsalary() {
		return esalary;
	}
	public void setEsalary(int esalary) {
		this.esalary = esalary;
	}
	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" + ename + ",
eaddress=" + eaddress + ", esalary=" + esalary + "]";
	}
}


package com.mainapp;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

import org.springframework.jdbc.core.JdbcTemplate;

public class Crud {

	private JdbcTemplate jdbcTemplate;  //inbuilt connection

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public void insert(int eid, String ename, String eaddress, int
esalary) {
```

```
            String sql="insert into
employee(eid,ename,eaddress,esalary) values(?,?,?,?)";
            int row =
jdbcTemplate.update(sql,eid,ename,eaddress,esalary);
            System.out.println(row);
      }

    public void update(int eid, String ename) {

            String sql="update employee set ename=? where eid=?";
            int row = jdbcTemplate.update(sql,ename,eid);
            System.out.println(row);
      }

    public void delete(int eid) {

            String sql="delete from employee where eid=?";
            int row = jdbcTemplate.update(sql,eid);
            System.out.println(row);
      }

    public void readAll() {

      String sql="select * from employee";
      List<Map<String, Object>> queryForList =
jdbcTemplate.queryForList(sql);
      //System.out.println(queryForList);

      for(Map<String, Object>  map     :  queryForList) {
            for( Entry<String, Object> entry :  map.entrySet() ) {
                  System.out.println(entry.getKey()+" :
"+entry.getValue());
            }
            System.out.println();
      }
    }

    public void readAllPojoBased() {

      String sql="select * from employee";
      List<Employee> list = jdbcTemplate.query(sql,new
RowMapperImpl());
      for(Employee emp: list) {
            System.out.println(emp);
      }

    }
```

```
    public void readSingleData(int eid) {

      String sql="select * from employee where eid=?";
      Employee emp = jdbcTemplate.queryForObject(sql,new
RowMapperImpl(),eid);
        System.out.println(emp);
    }
}


package com.mainapp;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class RowMapperImpl implements RowMapper<Employee> {

      @Override
      public Employee mapRow(ResultSet rs, int rowCount) throws
SQLException {

            Employee employee = new Employee();
            employee.setEid(rs.getInt("eid"));
            employee.setEname(rs.getString("ename"));
            employee.setEaddress(rs.getString("eaddress"));
            employee.setEsalary(rs.getInt("esalary"));

            System.out.println("ROW MAPPED"+ (rowCount+1));

            return employee;
      }

}
```

bean.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"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
```

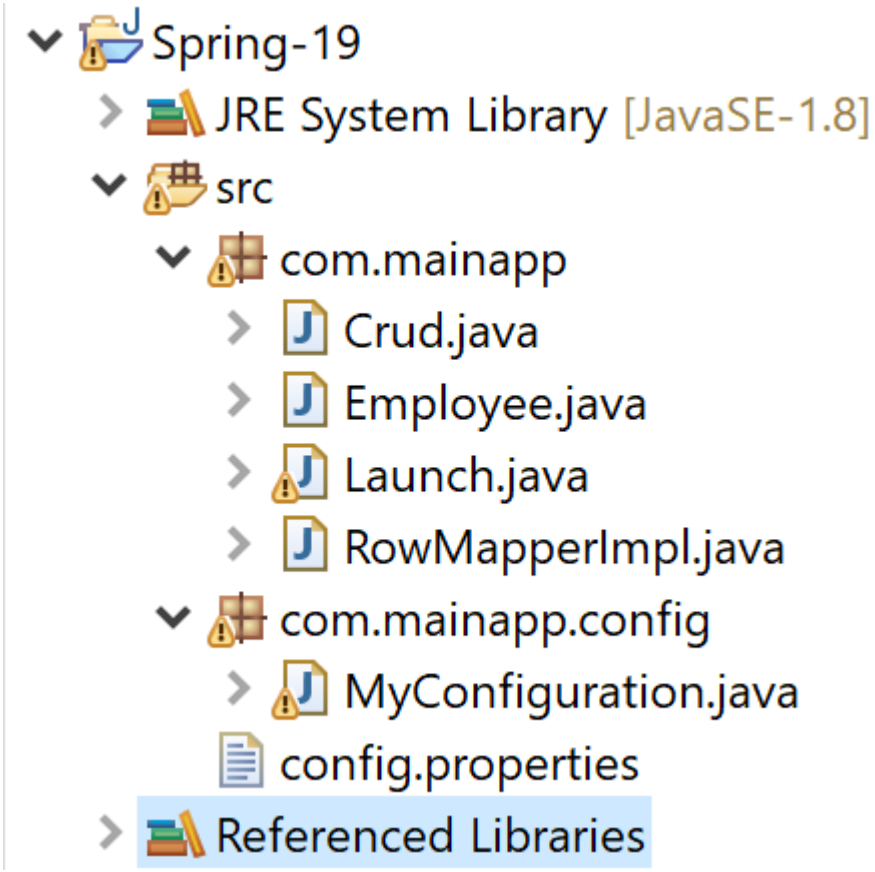
```
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd">

     <bean id="ds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource"
>
        <property name="url"
value="jdbc:mysql://localhost:3306/springjdbc"  ></property>
        <property name="username" value="root"  ></property>
        <property name="password" value=""  ></property>
        <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"  ></property>
     </bean>

     <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate" >
         <property name="dataSource" ref="ds"  ></property>
     </bean>

     <bean id="crud" class="com.mainapp.Crud" >
        <property name="jdbcTemplate" ref="jdbcTemplate"></property>

     </bean>

</beans>
```

==========

==========

==========

==========Program: Spring JDBC(Annotation Approach)==========




```
package com.mainapp;
import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;

import com.mainapp.config.MyConfiguration;
public class Launch {

	public static void main(String[] args) {

		AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(MyConfiguration.class);
		Crud bean = (Crud) ac.getBean("crud");
		//bean.insert(16,"raju","add1",1000);
		//bean.update(16, "kaju");
		//bean.delete(16);
		//bean.readAll();
		//bean.readAllPojoBased();
		bean.readSingleData(15);
	}
}
```

```
package com.mainapp;
public class Employee {
```

```
	private int eid;
	private String ename;
	private String eaddress;
	private int esalary;

	public int getEid() {
		return eid;
	}
	public void setEid(int eid) {
		this.eid = eid;
	}
	public String getEname() {
		return ename;
	}
	public void setEname(String ename) {
		this.ename = ename;
	}
	public String getEaddress() {
		return eaddress;
	}
	public void setEaddress(String eaddress) {
		this.eaddress = eaddress;
	}
	public int getEsalary() {
		return esalary;
	}
	public void setEsalary(int esalary) {
		this.esalary = esalary;
	}
	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" + ename + ",
eaddress=" + eaddress + ", esalary=" + esalary + "]";
	}
}
```

```
package com.mainapp;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

import org.springframework.jdbc.core.JdbcTemplate;

public class Crud {

	private JdbcTemplate jdbcTemplate;  //inbuilt connection
```

```
	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public void insert(int eid, String ename, String eaddress, int
esalary) {

		String sql="insert into
employee(eid,ename,eaddress,esalary) values(?,?,?,?)";
		int row =
jdbcTemplate.update(sql,eid,ename,eaddress,esalary);
		System.out.println(row);
	}

    public void update(int eid, String ename) {

		String sql="update employee set ename=? where eid=?";
		int row = jdbcTemplate.update(sql,ename,eid);
		System.out.println(row);
	}

    public void delete(int eid) {

		String sql="delete from employee where eid=?";
		int row = jdbcTemplate.update(sql,eid);
		System.out.println(row);
	}

    public void readAll() {

	  String sql="select * from employee";
	  List<Map<String, Object>> queryForList =
jdbcTemplate.queryForList(sql);
	  //System.out.println(queryForList);

	  for(Map<String, Object>  map     :  queryForList) {
		for( Entry<String, Object> entry :  map.entrySet() ) {
			System.out.println(entry.getKey()+" :
"+entry.getValue());
		}
		System.out.println();
	  }
    }

    public void readAllPojoBased() {

	  String sql="select * from employee";
```

```
      List<Employee> list = jdbcTemplate.query(sql,new
RowMapperImpl());
      for(Employee emp: list) {
            System.out.println(emp);
      }

    }

    public void readSingleData(int eid) {

      String sql="select * from employee where eid=?";
      Employee emp = jdbcTemplate.queryForObject(sql,new
RowMapperImpl(),eid);
         System.out.println(emp);
    }
}
```

```
package com.mainapp;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class RowMapperImpl implements RowMapper<Employee> {

      @Override
      public Employee mapRow(ResultSet rs, int rowCount) throws
SQLException {

            Employee employee = new Employee();
            employee.setEid(rs.getInt("eid"));
            employee.setEname(rs.getString("ename"));
            employee.setEaddress(rs.getString("eaddress"));
            employee.setEsalary(rs.getInt("esalary"));

            System.out.println("ROW MAPPED"+ (rowCount+1));

            return employee;
      }

}
```

```
package com.mainapp.config;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import com.mainapp.Crud;
@Configuration
@ComponentScan(basePackages = {"com.mainapp"})
@PropertySource("classpath:config.properties")
public class MyConfiguration {

	@Autowired
	private Environment environment;

	@Bean
	public DataSource dataSource() {

		DriverManagerDataSource driverManagerDataSource = new
DriverManagerDataSource();

driverManagerDataSource.setUrl(environment.getProperty("db.url"));

driverManagerDataSource.setUsername(environment.getProperty("db.user
name"));

driverManagerDataSource.setPassword(environment.getProperty("db.pass
word"));

driverManagerDataSource.setDriverClassName(environment.getProperty("
db.driver"));

		return driverManagerDataSource;
	}

	@Bean
	public JdbcTemplate jdbcTemplate() {
		return new JdbcTemplate(dataSource());
	}

	@Bean("crud")
	public Crud crud() {
		Crud crud = new Crud();
		crud.setJdbcTemplate(jdbcTemplate());
		return crud;
	}
}
```

config.properties

```
db.url=jdbc:mysql://localhost:3306/springjdbc
db.username=root
db.password=
db.driver=com.mysql.cj.jdbc.Driver
```

==========

==========

==========

Day-13

Spring MVC

What is Spring MVC?

Spring MVC is a web module of the Spring Framework used to build web applications based on the Model-View-Controller architecture.

It follows the MVC (Model-View-Controller) pattern, which means:

- **Model = your data and business logic**
- **View = what the user sees (like a webpage)**
- **Controller = handles user requests and connects model with view**

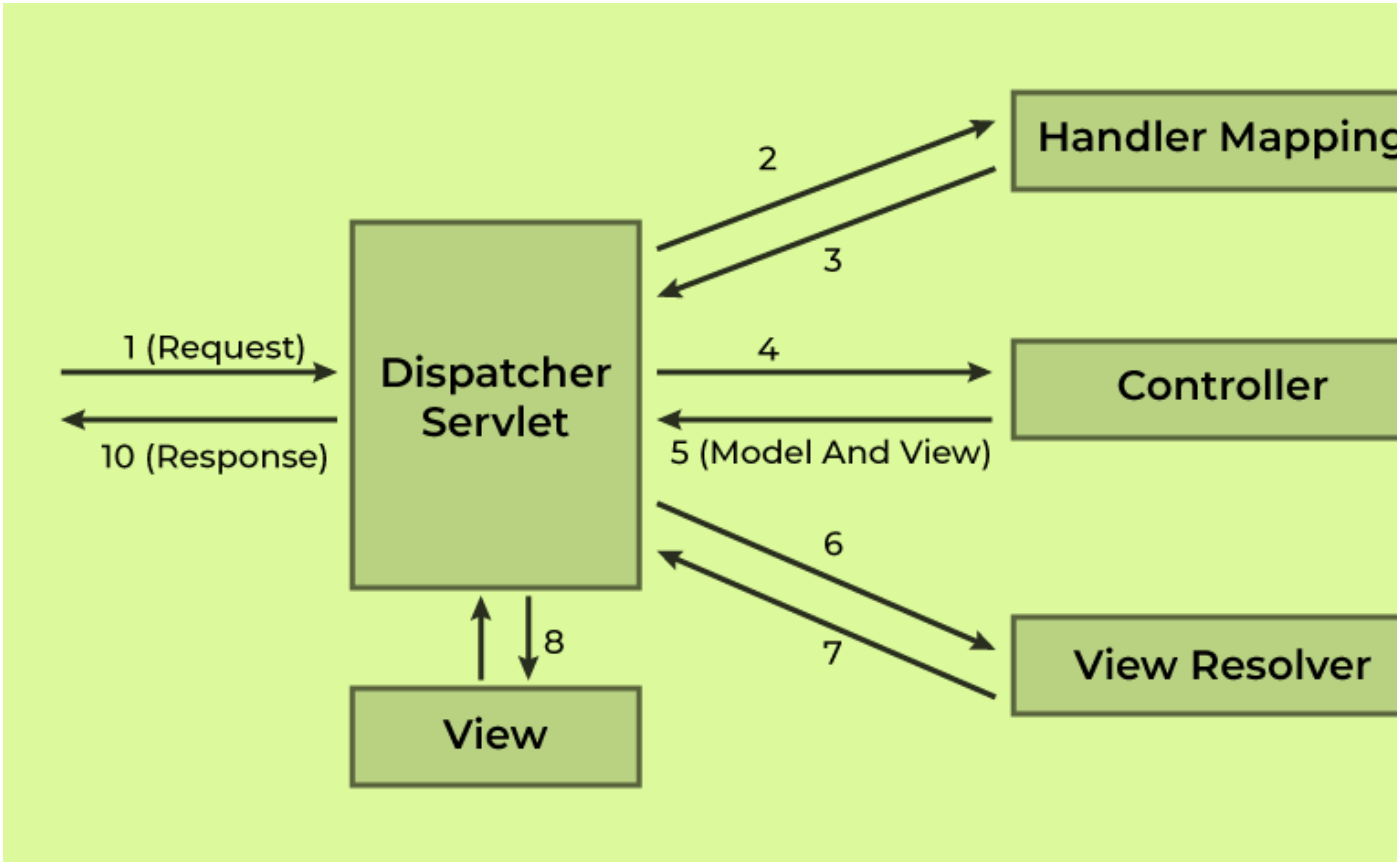
Spring MVC helps you:

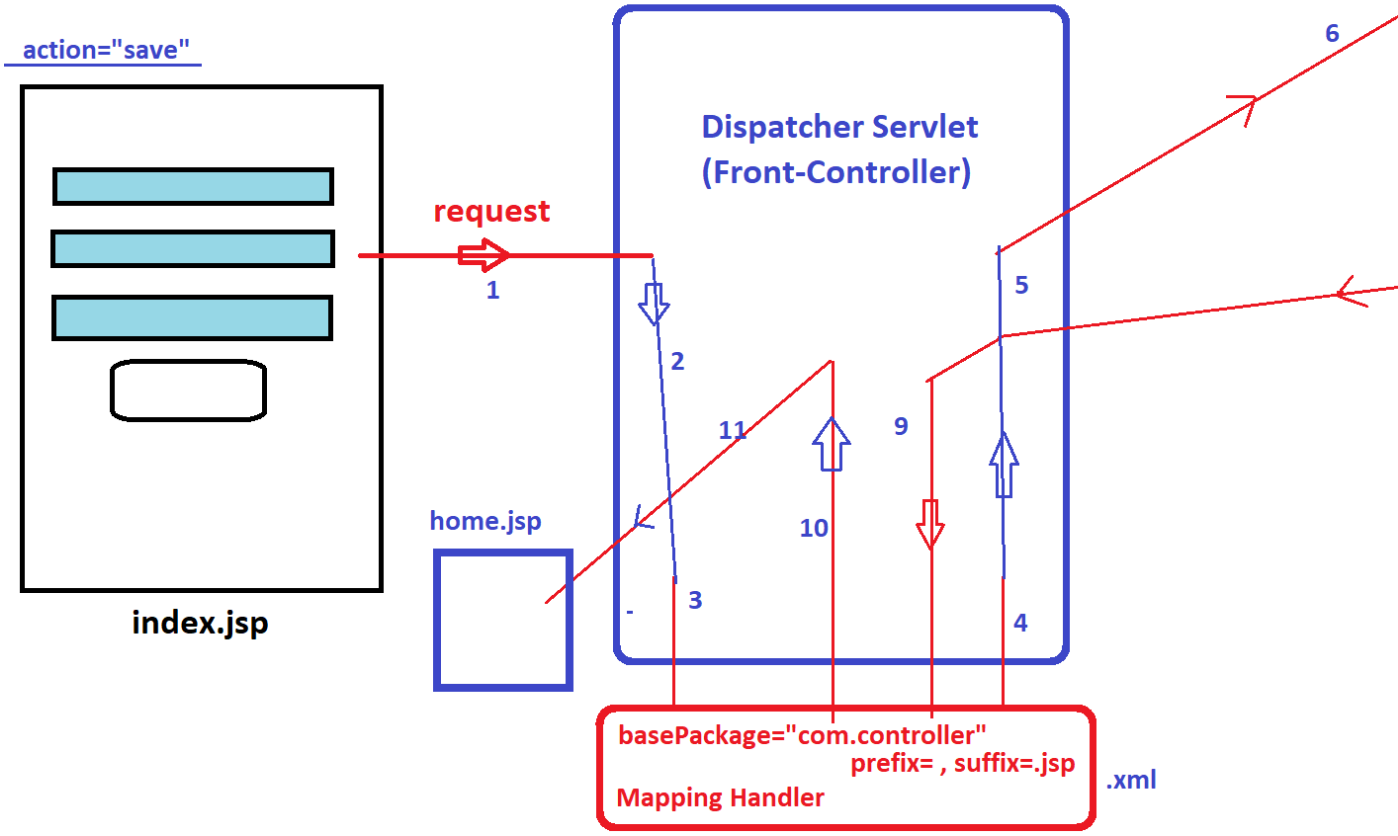
- **Keep code clean, seperate and organized**
- **Build dynamic web apps easily**
- **Handle user input, form data, and page navigation smartly**

Why Use Spring MVC?

- **It makes it easy to handle web requests & response.**
- **Helps separate logic, design, and routing (controller).**
- **Integrates smoothly with Spring Core (like Dependency Injection).**
- **Can work with JSP, Thymeleaf, React, or any view technology.**
- **Suitable for enterprise-grade web development.**

Flow of Spring MVC


Handler Mapping
2
3
1 (Request)
Dispatcher Servlet
4
Controller
10 (Response)
5 (Model And View)
6
7
8
View Resolver
View



Let's understand the flow with a simple request:

1. **User hits URL → /home**
2. **User sends request to a URL (e.g., localhost:8080/home)**
3. **DispatcherServlet receives it first (Front Controller in Spring MVC).**
4. **HandlerMapping finds which controller method should handle it.**

5. The matched @Controller method runs and interacts with Model (Java code).
6. It returns a View name.
7. ViewResolver maps the view name to the actual file (like home.jsp).
8. Response is rendered and sent back to the user.

What is DispatcherServlet?

- DispatcherServlet is the central controller in Spring MVC.
- It receives all incoming HTTP requests and delegates them to appropriate components like controllers, view resolvers, and handler mappings.
- It acts as the Front Controller in the Front Controller Design Pattern.

How it works (Workflow):

- Client sends request to a URL.
- DispatcherServlet receives the request.

- It uses a HandlerMapping to find the correct @Controller method.
- It executes the method and returns a logical view name (e.g., "home").
- It uses a ViewResolver to map the view name to a JSP (e.g., /WEB-INF/view/home.jsp).
- Finally, it renders the view back to the browser.

```
<servlet>
    <servlet-name>spring</servlet-name>

<servlet-class>org.springframework.web.servlet.
DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- The <url-pattern> tells Tomcat which URLs should go to Spring Dispatcher Servlet.

Handler Mapping in Spring MVC

What is HandlerMapping?

- It’s the component that maps incoming requests (like /login.do) to specific @Controller methods.

How does it work?

When a request comes in, DispatcherServlet asks the HandlerMapping:

- Which controller method should handle this URL?
- HandlerMapping checks annotations like @RequestMapping or @GetMapping and returns the correct method.
- It is also used to resolve view

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/be
ans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
tance"

xmlns:context="http://www.springframework.org/s
chema/context"

xmlns:mvc="http://www.springframework.org/schem
a/mvc"
    xsi:schemaLocation="

http://www.springframework.org/schema/beans
```

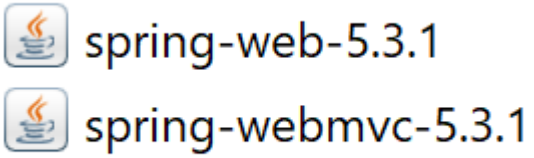
```
http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">


    <context:component-scan
        base-package="com.controller" />
    <bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix"
value="/WEB-INF/view/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

- **When you define a <servlet> and its <servlet-mapping> in web.xml, the handler mapping name must be the same as the servlet-name-servlet you've defined.**

Ex. if my Dispacther servlet name is spring then my Handler mapping name should spring –servlet.xml

Spring MVC Jars

To work with Spring MVC, in addition to the core Spring libraries (like spring-core, spring-context, etc.), you need to add two extra JARs:



In Spring MVC, there is a general rule:

Every dynamic page (like JSP with dynamic content) is typically called via a Controller.

Why is this the rule?

Spring MVC follows the Model-View-Controller (MVC) pattern:

- **Client (Browser) makes a request**
- **Controller receives and handles the request**

- It performs logic or calls service/DB
- Then returns the View (like JSP) with data

So, What Happens If You Try to Directly Call a Dynamic JSP?

You can access static or simple JSPs directly via URL (if you allow it).

- But in enterprise applications, direct access to a Dynamic JSPs is discouraged or even restricted (for security and architecture reasons).
- JSPs are usually kept in /WEB-INF folder so they can't be accessed directly from the browser — only through a controller.

What are Stereotype Annotations in Spring?

- Stereotype annotations in Spring are special annotations used to define Spring-managed components (beans) and specify their roles in an application such as controller, service, repository, etc.

- **These annotations help Spring automatically detect and register classes as beans during component scanning.**

Common Stereotype Annotations:

Annotation	Purpose	Layer
`@Component`	**Generic annotation for any Spring-managed component**	**General purpose**
`@Controller`	**Marks a class as a** Spring MVC Controller	**Presentation layer**
`@Service`	**Marks a class as a** service/business logic **component**	**Service layer**
`@Repository`	**Marks a class as a** DAO (data access object)	**Persistence layer**

@RequestMapping

- It is the parent/general annotation used to map HTTP requests to controller methods.
- It can be used at class level and method level.
- It supports all HTTP methods (GET, POST, PUT, DELETE, etc.)

```
@RequestMapping("/home")
public String homePage() {
    return "home";
}
```

- If you want to specify method type:

```
@RequestMapping(value = "/save", method = RequestMethod.POST)
public String saveData() {
    return "saved";
}
```

@GetMapping

- Shortcut for @RequestMapping(method = RequestMethod.GET)
- Used to handle HTTP GET requests.

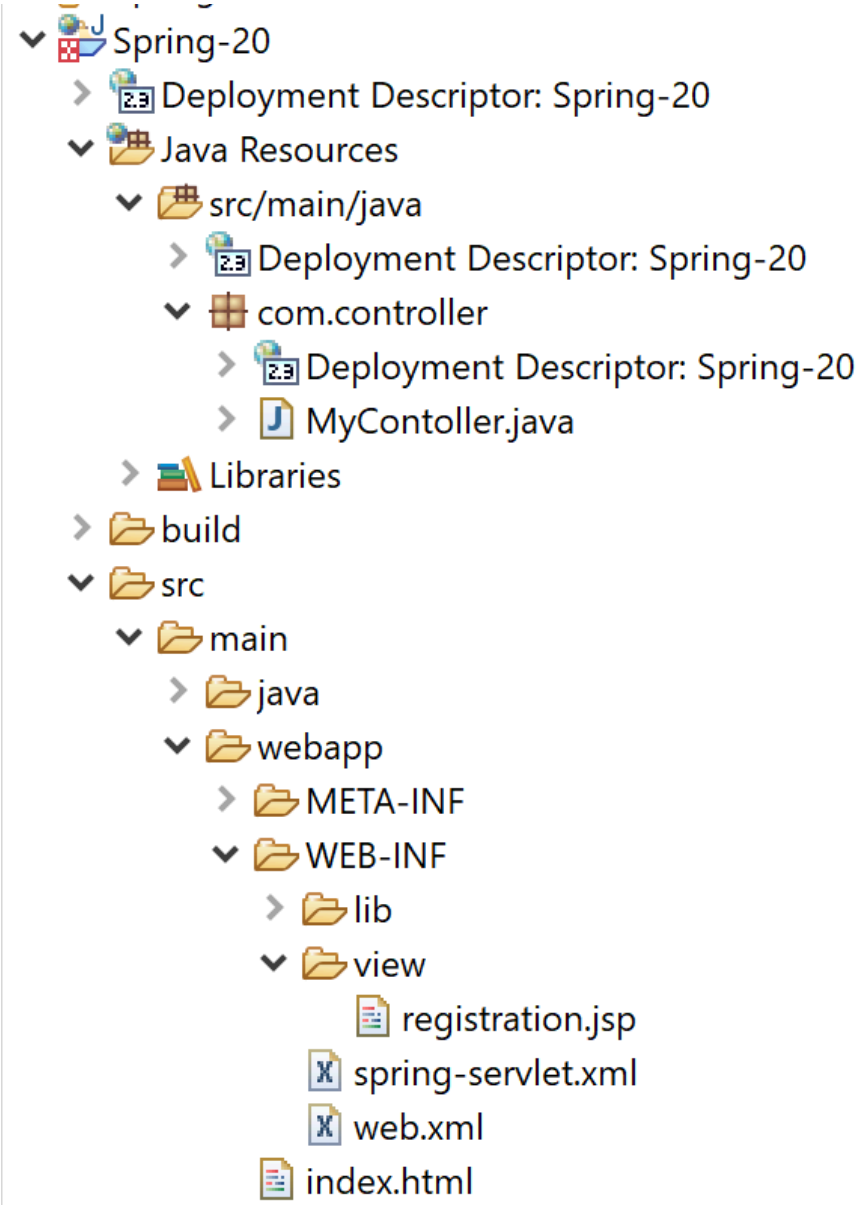
```
@GetMapping("/register")
public String showForm() {
    return "registerForm";
}
```

@PostMapping

- **Shortcut for @RequestMapping(method = RequestMethod.POST)**
- **Used to handle HTTP POST requests, usually for form submission or saving data.**

```
@PostMapping("/submitForm")
public String submit(@ModelAttribute User user) {
    return "success";
}
```

==========Program: Spring MVC(Annotation Approach)==========




```java
package com.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/admission")
public class MyContoller {

	@GetMapping("/registration.do")
	//@RequestMapping(value =  "/registration.do" , method =
RequestMethod.GET  )
	public String register() {
```

```
            System.out.println("TEST....");
            return "registration";

      }

}
```

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Index</title>
</head>
<body>

   <a href="admission/registration.do" >REGISTRATION</a>

</body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
      <display-name>Spring-20</display-name>


      <servlet>
         <servlet-name>spring</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</se
rvlet-class>
      </servlet>

      <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>*.do</url-pattern>
      </servlet-mapping>
```

```
	<welcome-file-list>
		<welcome-file>index.html</welcome-file>
		<welcome-file>index.htm</welcome-file>
		<welcome-file>index.jsp</welcome-file>
		<welcome-file>default.html</welcome-file>
		<welcome-file>default.htm</welcome-file>
		<welcome-file>default.jsp</welcome-file>
	</welcome-file-list>
</web-app>
```

spring-servlet.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"
	xmlns:context="http://www.springframework.org/schema/context"
	xmlns:mvc="http://www.springframework.org/schema/mvc"
	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
	  http://www.springframework.org/schema/mvc
	  http://www.springframework.org/schema/mvc/spring-mvc.xsd">

	  <context:component-scan
base-package="com.controller"></context:component-scan>

	  <bean
class="org.springframework.web.servlet.view.InternalResourceViewReso
lver" >
		<property name="prefix" value="/WEB-INF/view/"
></property>
		<property name="suffix" value=".jsp" ></property>
	  </bean>

	  <!-- /WEB-INF/view/registration.jsp -->

</beans>
```

registration.jsp

```
<%@ page language="java" contentType="text/html;
charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Registration</title>
</head>
<body>
<h1>REGISTRATION PAGE</h1>
</body>
</html>
```

==========

==========

==========

Day-14

Spring MVC

Spring MVC: Sending Data to Controller

1. HttpServletRequest (RARE CASE)

- Directly read form data using request.getParameter().
- Not recommended for most use cases — low level and verbose.

```
@PostMapping(path = {"insert"})
public String insertData(HttpServletRequest request) {
    String id = request.getParameter("id");
    String name = request.getParameter("name");
    System.out.println(id + " - " + name);
    return "insert";
}
```

2. @ModelAttribute (MOST COMMON)

- Automatically maps form fields to Java object properties.
- Requires a DTO (Data Transfer Object) or model class like EmployeeDTO.
- Clean and preferred method in Spring MVC.

```
@PostMapping(path = {"insert"})
public String insertData(@ModelAttribute EmployeeDTO employeeDTO) {
    System.out.println(employeeDTO.getId() + " - " + employeeDTO.getName());
```

```
        return "insert";
    }
```

- Spring automatically binds the form fields to the fields in the EmployeeDTO class.

3. @RequestParam (SPECIFIC CASES)

- Use when you want to extract only a few specific parameters.
- Useful for handling simple values like Strings, integers, etc.
- Automatically handles type conversion from String (in the URL) to the specified Java type (e.g., int, double, etc.)
- Used to extract query parameters from the URL (the part after ?).

```
@GetMapping("delete")
public String insertData(@RequestParam("id")
int id) {
    System.out.println(id);
    return "insert";
}
```

Understanding the Model

Model is an object that holds the data in the application and is used to transfer data between Controller and View.

It can be:

- **A DTO (like EmployeeDTO)**
- **An Entity class**
- **Spring's Model interface (org.springframework.ui.Model)**

Model:

- **General term for any class that holds application data.**

Detailed Breakdown:

Term	Full Form	Purpose	Can act as Model?
Entity		Represents DB table (JPA). Used for persistence.	✅ Yes

- **DTO Data Transfer Object**
  Carries data between layers (e.g., Controller ↔ View). ✅ Yes
- **POJO Plain Old Java Object**
  Any simple Java class without framework-specific code. ✅ Yes

❗ Problem: Resubmission on Page Refresh

Why It Happens?

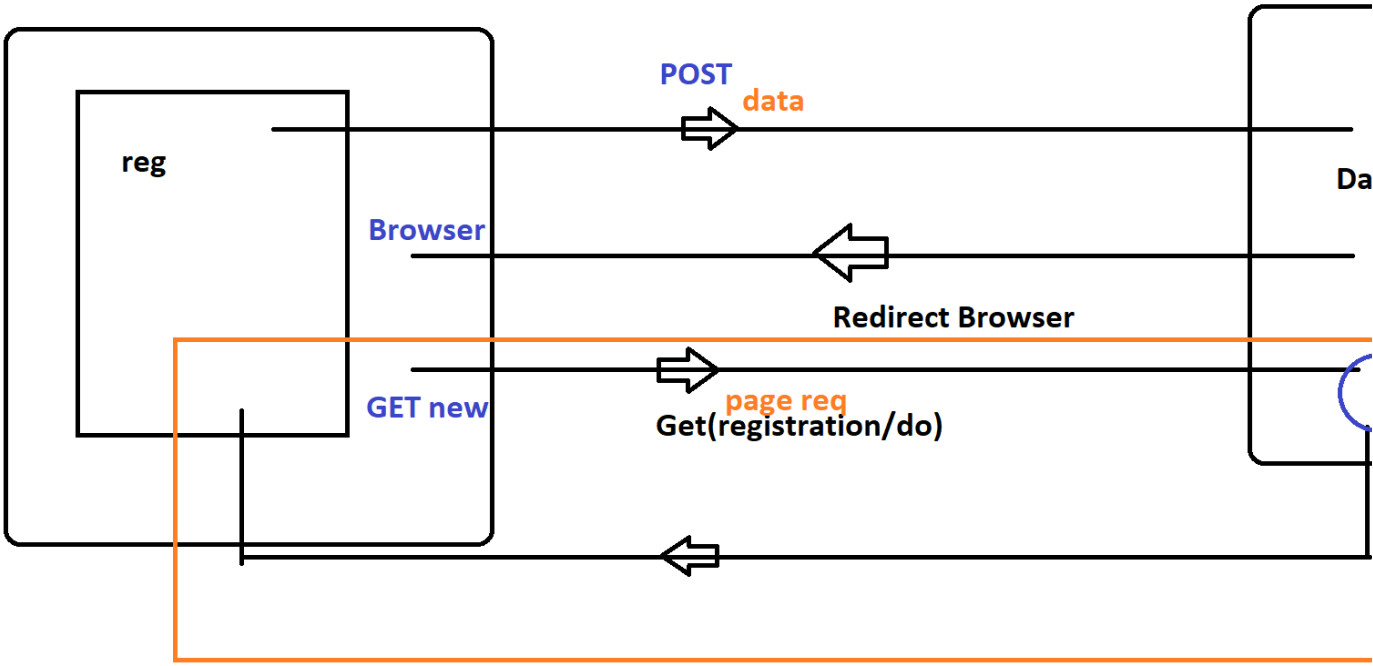
After form submission using POST, if the user refreshes the page, the browser resend the POST request, leading to duplicate submissions.

✅ Solution: PRG (Post-Redirect-Get) Pattern

How PRG Works:

- **User submits a POST request to the controller.**

- Controller processes the data.
- Instead of returning a view directly, controller redirects to browser and generate a new GET request.
- Browser sends the new GET request, showing the view without resubmitting data.



✅ **Code Example:**

```
@PostMapping("/insert")
public String insertData(@ModelAttribute
EmployeeDTO employeeDTO) {
    System.out.println(employeeDTO.getId() + "
- " + employeeDTO.getName());
    // Redirecting to avoid resubmission
    return "redirect:/insert";
```

```
}
@GetMapping("/insert")
public String insertView() {
    return "insert";
}
```

- **Always use redirect:/... when handling form submission to prevent duplicate submissions on refresh.**

Day-15

Spring MVC

@PathVariable

✅ Purpose:

- **Used to extract data from the URL path itself (the part inside {}).**

```
@GetMapping("/register/{id}")
```

```
    public String register(@PathVariable int id) {
        System.out.println("ID = " + id);
        return "success";
    }
```

✅ **URL Example:**

/register/101

What is Context Path in Spring MVC?

- **In a Java web application, including those built with Spring MVC, the context path is the base path of your application when it is deployed on a server.**
- **The context path is the portion of the URL that represents the root of your web application.**
- **For example, if your app is deployed as:**
  - **http://localhost:8080/myapp**
  - **Then /myapp is the context path.**

Using Context Path in JSP:

- In JSP files, use the pageContext object to automatically get the context path:

```
<form
action="${pageContext.request.contextPath
}/insert" method="post">
    <!-- your form fields here -->
</form>
```

- This ensures that no matter where your application is deployed (like /myapp, /adminpanel, etc.), the form will submit to the correct path.

Day-16

Model Object

- Model is used to send data from controller to view (JSP, Thymeleaf, etc.) in the same request.

```
@GetMapping("/hello")
public String helloPage(Model model) {
    model.addAttribute("name", "xyz");
```

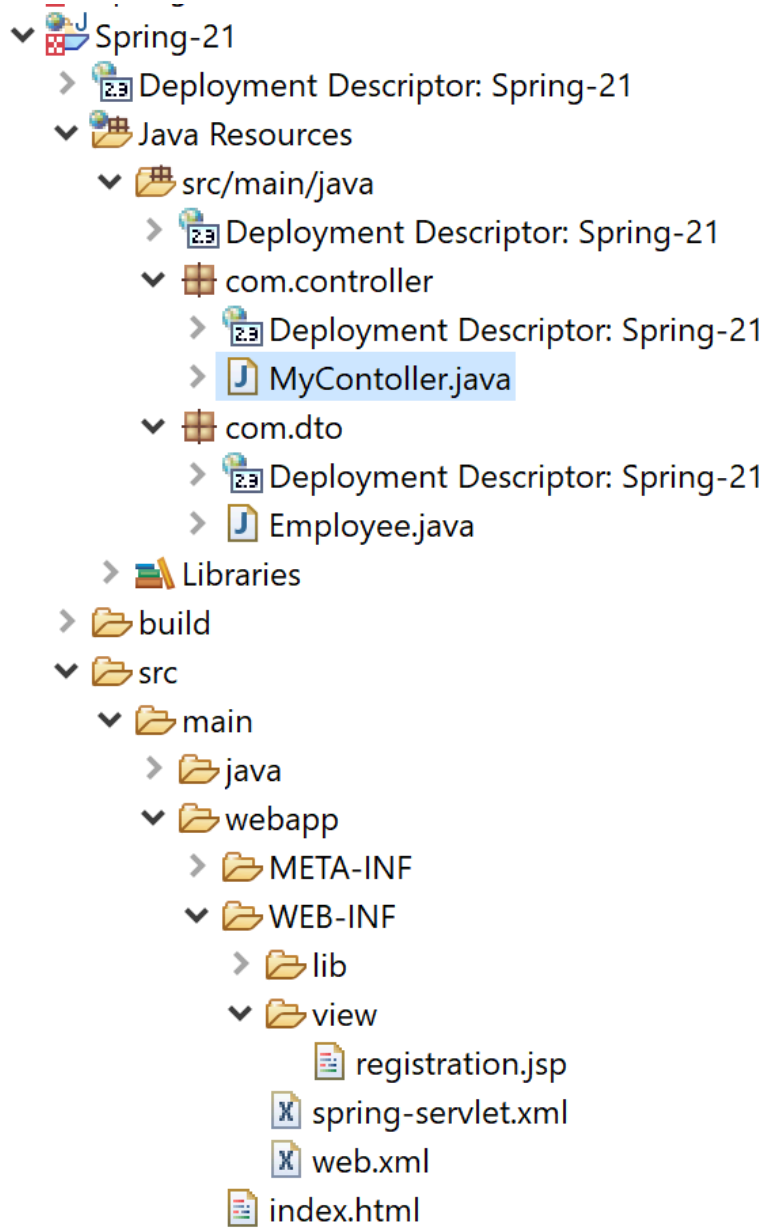
```
    return "hello"; // returns hello.jsp
}
```

RedirectAttributes

- When you redirect, the request is new, so Model won’t retain data.
- Use RedirectAttributes to pass data only during redirect.

```
@PostMapping("/save")
public String saveData(RedirectAttributes
redirectAttributes) {

redirectAttributes.addFlashAttribute("message",
"Data saved successfully!");
    return "redirect:/success";
}

@GetMapping("/success")
public String successPage(Model model) {
    // model will contain flash message
    return "success";
}
```

==========Program: Spring MVC(Annotation Approach)==========



index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Index</title>
</head>
<body>

   <a href="dynamic/admission/registration/">REGISTRATION</a>

   <a href="dynamic/admission/delete/?id=10&name=raju" >TEST</a>

</body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
      <display-name>Spring-21</display-name>

      <servlet>
         <servlet-name>spring</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</se
rvlet-class>
      </servlet>

      <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/dynamic/*</url-pattern>
      </servlet-mapping>

      <welcome-file-list>
            <welcome-file>index.html</welcome-file>
            <welcome-file>index.htm</welcome-file>
            <welcome-file>index.jsp</welcome-file>
            <welcome-file>default.html</welcome-file>
            <welcome-file>default.htm</welcome-file>
            <welcome-file>default.jsp</welcome-file>
      </welcome-file-list>
```

```
</web-app>
```

spring-servlet.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"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:mvc="http://www.springframework.org/schema/mvc"
      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
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

        <context:component-scan
base-package="com.controller"></context:component-scan>

        <bean
class="org.springframework.web.servlet.view.InternalResourceViewReso
lver" >
            <property name="prefix" value="/WEB-INF/view/"
></property>
            <property name="suffix" value=".jsp" ></property>
        </bean>

        <!-- /WEB-INF/view/registration.jsp -->

</beans>
```

registration.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8" isELIgnored="false"  %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Registration</title>
</head>
```

```
<body>
    ${msg}
      <h1>REGISTRATION PAGE</h1>

      <form
action="${pageContext.request.contextPath}/dynamic/admission/registr
ation"  method="post">
           <input type="text" name="id" placeholder="enter id"><br>
           <input type="text" name="name" placeholder="enter
name"><br>
           <input type="text" name="address" placeholder="enter
address"><br>
           <input type="text" name="salary" placeholder="enter
salary"><br>
           <button>save</button>
      </form>

</body>
</html>
```

```
package com.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;


import com.dto.Employee;

@Controller

@RequestMapping("/admission")

public class MyContoller {




     @GetMapping("registration")
      public String registerView(Model model) {
           if(!model.containsAttribute("message")) {
              model.addAttribute("message","this is my registration
page");
```

```
            }
            return "registration";
      }

      @PostMapping("registration")
      public String register( @ModelAttribute Employee employee,
RedirectAttributes redirectAttributes ) {

            System.out.println(employee);
            redirectAttributes.addFlashAttribute("message","data
inserted");
            return "redirect:/dynamic/admission/registration";
      }


      @GetMapping("delete")
      public String delete( @RequestParam("id") int id,
@RequestParam("name") String name) {

            System.out.println(id);
            System.out.println(name);
            return "redirect:/index.html";
      }

      @GetMapping("deletebyid/{id}")  //dispatcherServlet URL= do
not use .
      public String delete(@PathVariable int id) {
            System.out.println(id);
            return "redirect:/index.html";
      }
}



package com.dto;

public class Employee {

      private String id;
      private String name;
      private String address;
      private String salary;

      public Employee() {
            // TODO Auto-generated constructor stub
      }

      public Employee(String id, String name, String address, String
salary) {
            super();
```

```
		this.id = id;
		this.name = name;
		this.address = address;
		this.salary = salary;
	}

	public String getId() {
		return id;
	}

	public void setId(String id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getAddress() {
		return address;
	}

	public void setAddress(String address) {
		this.address = address;
	}

	public String getSalary() {
		return salary;
	}

	public void setSalary(String salary) {
		this.salary = salary;
	}

	@Override
	public String toString() {
		return "Employee [id=" + id + ", name=" + name + ",
address=" + address + ", salary=" + salary + "]";
	}
}
```

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

SpringBoot

- Spring Boot is a tool/project developed to simplify Java development, especially for Spring-based applications.
- It is built on top of the Spring Framework, making it an extension of Spring.
- Developed by: Pivotal Software (now a part of VMware)
- Initial Release: April 2014
- Purpose: Rapid and simplified development of Spring applications.

✅ Key Features

Convention Over Configuration

→ **Reduces boilerplate code through sensible defaults.**

No XML Configuration Required

→ **Everything is managed via application.properties or application.yml**

Spring Boot Starter POMs

→ **Auto-downloads required jars using Maven (internally uses Maven tool)**

Embedded Web Server

→ **Comes with Tomcat (default), Jetty, or Undertow on port 8080**

Opinionated Defaults

→ **Auto-configures beans and settings for rapid development.**

Framework of Frameworks

→ **Combines Spring Core, MVC, JPA, AOP, Security, etc., under one umbrella.**

Supports RAD (Rapid Application Development)

Prerequisites

- Core Java & Advanced Java
- Hibernate
- Spring Core & Spring MVC

Maven Tool Overview

- Maven is a build automation and dependency management tool.
- Uses pom.xml to define project structure, dependencies, and plugins.
- Downloads jars from remote repository like https://mvnrepository.com
- Stores them in local repository: C:\Users\<username>\.m2\repository

Example Dependency (MySQL)

```
<dependencies>
  <dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
```

```
    <version>8.3.0</version>
  </dependency>
</dependencies>
```

How to Create a Spring Boot Project

- To create a Spring Boot project, we will use STS (Spring Tool Suite).
- STS is a ready-to-use IDE specially made for Spring development.
- You can also use Spring Initializr (a web-based project generator) if you want to quickly create a Spring Boot project and import it into any IDE (like Eclipse or IntelliJ).

Day-17

How to Interact with Database

- Spring Boot makes database interaction much easier compared to plain JDBC.
- To interact with a database in Spring Boot, it provides a very powerful concept called Spring Data JPA.
- This module is built on top of JPA (Java Persistence API) and makes database operations extremely easy by removing the need for most boilerplate code.
- With Spring Data JPA, you can perform CRUD (Create, Read, Update, Delete) operations, pagination, sorting, and even complex queries — often without writing a single line of SQL.

JPA, Hibernate, and Spring Data JPA

1. JPA (Java Persistence API)

Type: Specification (not an implementation).

Purpose: Defines a set of interfaces and annotations for ORM (Object Relational Mapping) operations.

2. Hibernate

Type: ORM Framework (Implementation of JPA + its own native API).

Purpose:

- **Implements JPA → Makes JPA interfaces work.**
- **Provides its own native API (more powerful and feature-rich than standard JPA).**

3.Spring Boot and JPA

In Spring Boot, JPA is used as a specification.

By default, Spring Boot uses Hibernate as the JPA implementation.

But, Without extra help, using JPA + Hibernate means:

- **Manual repository creation.**
- **Manual method writing.**
- **Manual wiring.**

4. Spring Data JPA

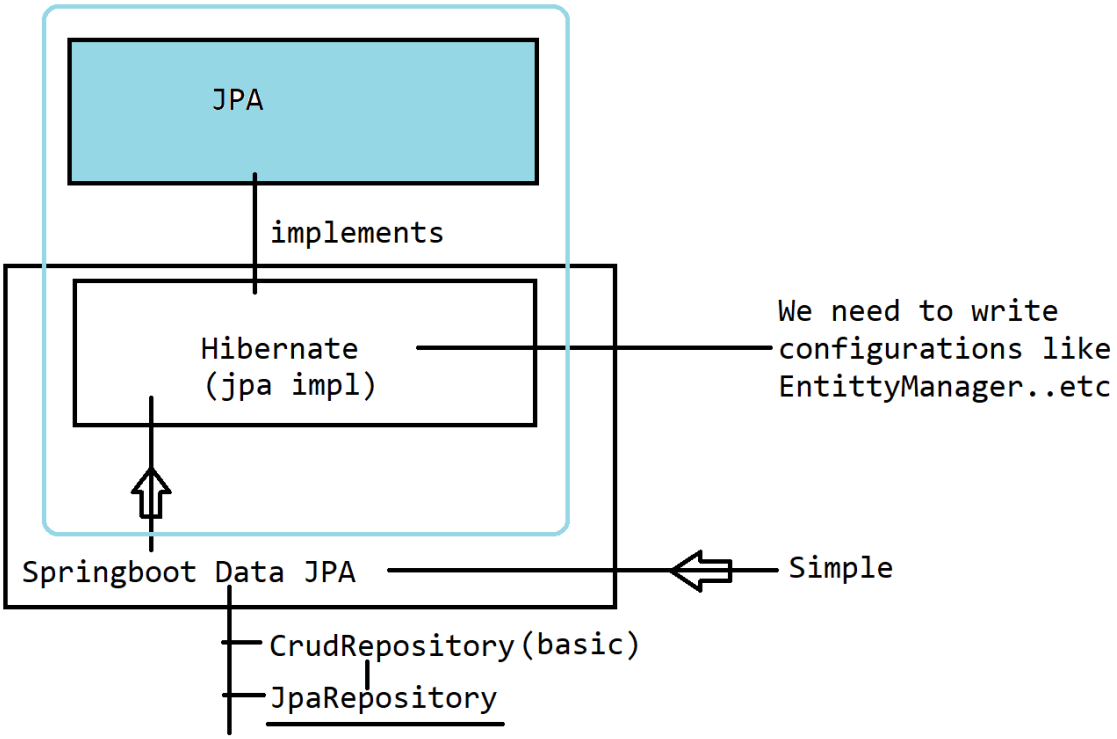
Type: Part of the Spring Data family.

Purpose: Simplifies JPA usage in Spring Boot by removing boilerplate code.

Works on Top of: JPA → Hibernate.

Key Features:

- **Automatically creates repository implementations at runtime.**
- **No need to write SQL for most CRUD operations.**
- **Supports method-name-based queries.**



Main Interfaces:

- **CrudRepository<T, ID>**
- **JpaRepository<T, ID>**

Note:

In Spring Boot, when we create an interface like MyRepo and extend JpaRepository, we don't write any code for methods such as findById() — we only declare that we want them. When the application

starts, Spring automatically creates a special object for our interface that already has the working code inside

Difference Between CrudRepository and JpaRepository

CrudRepository

- Basic database operations: save(), findById(), findAll(), delete()
- Good for simple CRUD work.
- Less features.

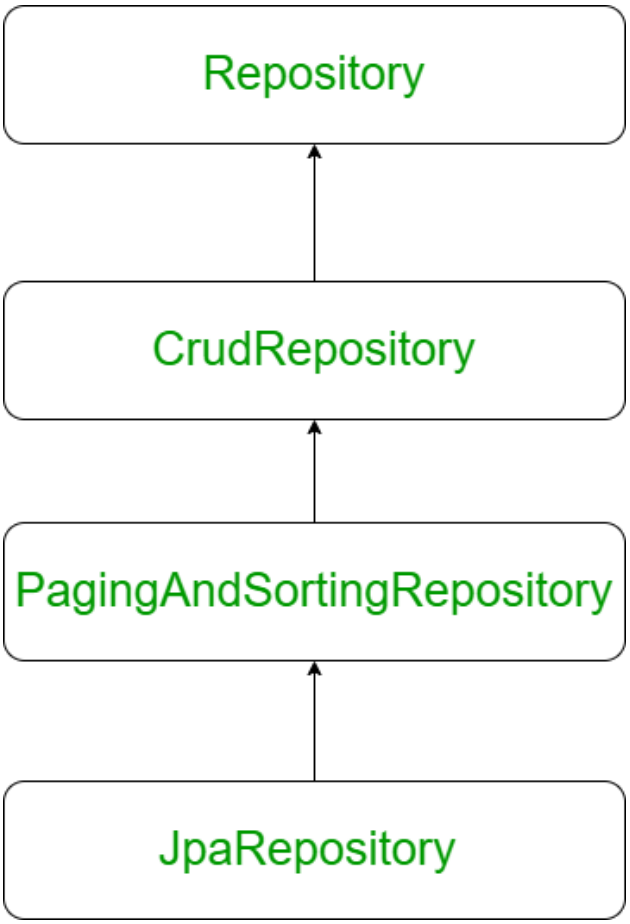
JpaRepository

- Includes everything from CrudRepository.
- Adds extra JPA-related features like:
    1. Pagination (findAll(Pageable p))
    2. Sorting (findAll(Sort s))
    3. Batch operations.

- **More powerful and used in most Spring Boot projects.**

In short:

- **JpaRepository = CrudRepository + Extra features for pagination, sorting, and advanced JPA support.**



Spring Data JPA automatically implements repository

methods if you follow a specific naming convention.

Methods like findByFieldNameContaining are called Derived Query Methods, and Spring Boot will automatically generate their implementation at runtime, so you don't have to write custom queries manually.

Common Keywords in Spring Data JPA Method Names

These keywords are used after your entity field name in method names, and Spring Data JPA will create the query automatically.

Keyword Example Method Generated Query (Assume Entity: User)

- **findBy findByName(String name) SELECT u FROM User u WHERE u.name = ?1**
- **findBy…And… findByNameAndAge(String name, int age) SELECT u FROM User u WHERE u.name = ?1 AND u.age = ?2**
- **findBy…Or… findByNameOrCity(String name, String city) SELECT u FROM User u WHERE u.name = ?1 OR u.city = ?2**

- **Containing findByNameContaining(String keyword) SELECT u FROM User u WHERE u.name LIKE %?1%**
- **StartsWith findByNameStartingWith(String prefix) SELECT u FROM User u WHERE u.name LIKE ?1%**
- **EndsWith findByNameEndingWith(String suffix) SELECT u FROM User u WHERE u.name LIKE %?1**
- **Like findByNameLike(String pattern) SELECT u FROM User u WHERE u.name LIKE ?1 (you pass % manually)**
- **GreaterThan findByAgeGreaterThan(int age) SELECT u FROM User u WHERE u.age > ?1**
- **LessThan findByAgeLessThan(int age) SELECT u FROM User u WHERE u.age < ?1**
- **Between findByAgeBetween(int start, int end) SELECT u FROM User u WHERE u.age BETWEEN ?1 AND ?2**

SpringApplication.run(YourProjectName.class, args); is the starting point.

When you call it:

- **It starts the Spring Boot framework.**

- Creates the Spring Application Context (container where all beans/objects are managed).
- Runs auto-configuration (sets up beans based on dependencies).
- Scans your project for @Component, @Service, @Repository, etc., and creates their objects.
- Finally, it keeps your app running (for web apps, it starts the embedded Tomcat server).
- Without it, Spring Boot won't start — your program will just be a normal Java program (like in your example where it only prints "test....").
- ConfigurableApplicationContext → Think of it as Spring's big container interface that holds all your beans and can be started, stopped, and refreshed.

Lombok in Spring Boot

- Lombok is a Java library that helps you write less code by automatically generating common methods like getters, setters, constructors, toString(), etc., at compile time.

Common Lombok Annotations

@Getter → Creates getter methods for all fields.

@Setter → Creates setter methods for all fields.

@Data → Creates getter, setter, toString(), equals(), hashCode(), and required constructor for final fields.

@NoArgsConstructor → Creates a no-argument constructor.

@AllArgsConstructor → Creates a constructor with all fields as parameters.

@Builder → Enables builder pattern for object creation.

Day-18

Spring Boot MVC – Building Web Applications with JSP

1. Introduction to Spring Boot MVC

Spring Boot is a framework that simplifies the creation of production-ready Spring applications. When building web applications, Spring Boot follows the MVC (Model–View–Controller) pattern.

2. View Technologies in Spring Boot

Spring Boot supports multiple view/template rendering approaches. Some are server-side like JSP & Thymeleaf, while others, like React, are client-side frameworks.

In this session, we will learn how to create a JSP-based Spring Boot web application from scratch.

We will cover:

- **Setting up the project with all required JSP dependencies and plugins.**
- **Understanding the MVC flow in Spring Boot using JSP as the view technology.**
- **Organizing files into static and dynamic resources (webapp, WEB-INF, views).**
- **Writing a Model class with Lombok**

- Creating Controllers to handle HTTP requests and bind form data.
- Designing JSP pages

Why We Need Tomcat Jasper in JSP-based Spring Boot Projects

1. What is Tomcat Jasper?

- Jasper is the JSP engine inside Apache Tomcat.
- Its job is to translate JSP files into Java servlets and then compile those servlets into .class files so they can run on the JVM.
- Without Jasper, JSP files cannot be processed — Spring Boot will just send them as plain text instead of rendering HTML.

2. Why Add tomcat-embed-jasper in Spring Boot?

- Spring Boot uses embedded Tomcat by default.
- The embedded Tomcat doesn’t include the Jasper engine by default.

By adding:

```
<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

we are telling Spring Boot:

"Enable JSP compilation in the embedded Tomcat server."

Day-19

Spring Boot Project

In this session, we will start our Spring Boot web project

Download the template

- **Get a HTML/CSS/JS template for the project.**

Configure the template inside Spring Boot

- Place all static resources (CSS, JS, images) inside src/main/webapp so they are accessible publicly.
- Place all JSP pages inside src/main/webapp/WEB-INF/view so they can be rendered via Controller.

Using ${pageContext.request.contextPath}

- Add ${pageContext.request.contextPath} before every CSS, JS, and image path in JSP to make them work regardless of the app context path.

Creating Controller to view all pages

- Create a Spring Boot controller that returns JSP views for:
  1. Home (/)
  2. About
  3. Services
  4. Contact