SPRING

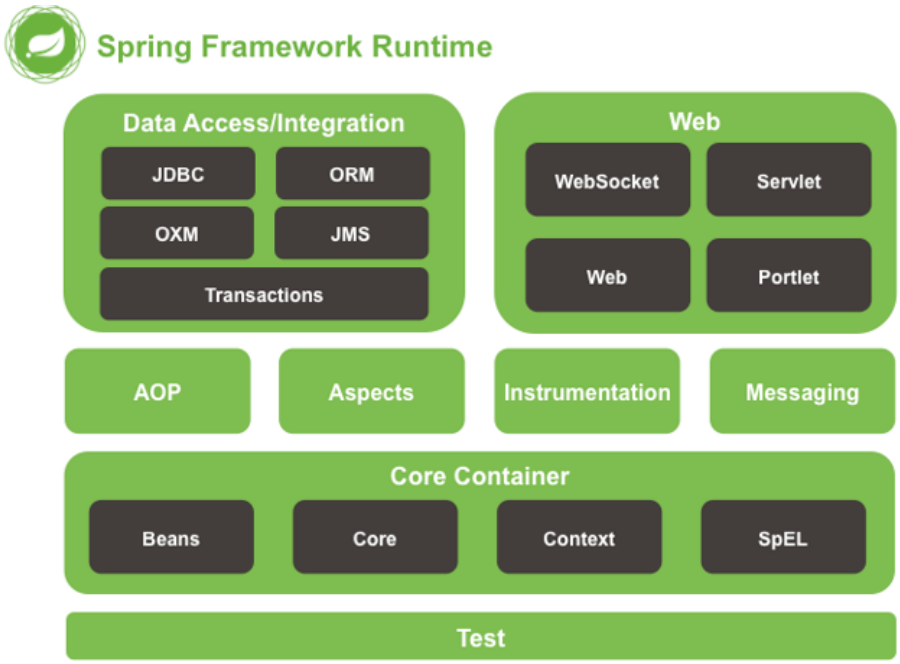
Framework

SATYA KAVETI

SATYACODES.COM

Contents

I.SpringCore



IoC Container

Dependency Injection((Design Pattern))

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case **we provide the information from the external source such as XML file**. It makes our code loosely coupled and easier for testing.

```
class Employee {
        Address address;
        Employee(Address address) {
                this.address = address;
        }
        public void setAddress(Address address) {
                this.address = address;
        }
}
```

In above program, instance of Address class is provided by **external source such as XML file either by constructor or setter method.**

IOC Container (Inversion of Control)

In Spring framework, **IOC container is responsible for Dependency Injection**. We provide meta-data to the IOC container either by XML file or annotation.

The IoC container is responsible to instantiate, configure and assemble the objects. **The IoC container gets information from the XML file** and works accordingly.it will perform below tasks.

- instantiate the application class
- configure the object
- assemble the dependencies between the objects

There are two types of IoC containers.

1. **BeanFactory**
2. **ApplicationContext**

1.BeanFactory

- The BeanFactory interface provides a basic configuration mechanism capable of managing any type of object.
- It only supports Bean **instantiation/wiring**
- XmlBeanFactory is the implementation class

```
Resource resource=new ClassPathResource("applicationContext.xml");
BeanFactory factory=new XmlBeanFactory(resource);
```

2.ApplicationContext

- **ApplicationContext** is a sub-interface of **BeanFactory**. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the **WebApplicationContext** for use in web applications.

- In short, the BeanFactory provides the configuration framework and basic functionality, and the **ApplicationContext adds more enterprise-specific functionality.**

The most commonly used `ApplicationContext` implementations are:

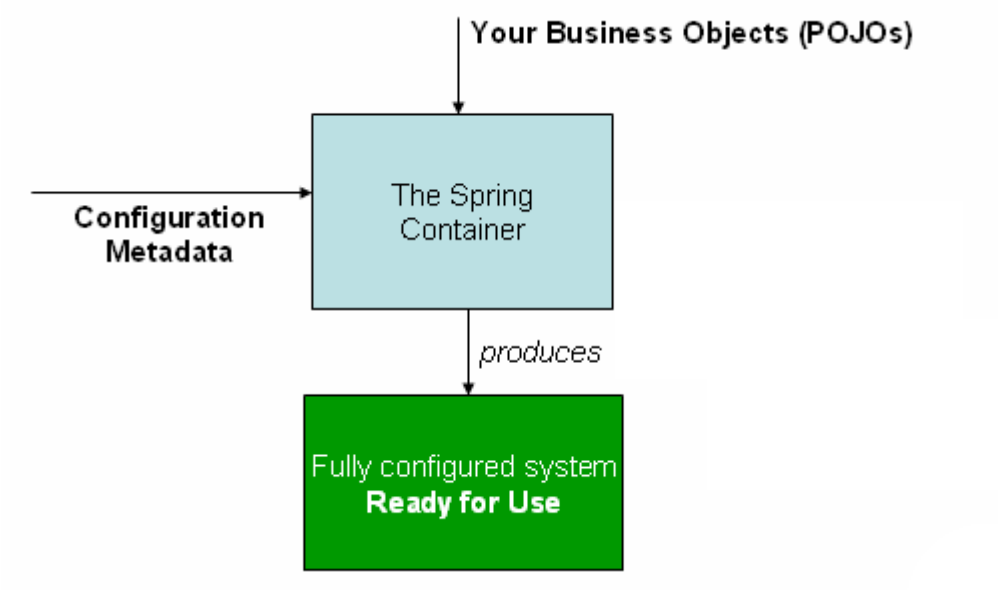
- **FileSystemXmlApplicationContext** Here you need to provide the full path of the XML bean configuration file to the constructor.
- **ClassPathXmlApplicationContext** Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
- **WebXmlApplicationContext** – This container loads the XML file with definitions of all beans from within a web application.

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
```

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Automatic BeanPostProcessorregistration	No	Yes
Automatic BeanFactoryPostProcessorregistration	No	Yes
Convenient MessageSource access (for i18n)	No	Yes
ApplicationEvent publication	No	Yes

Configuration metadata

Your application classes are combined with configuration metadata so that after the **ApplicationContext** is created and initialized, you have a fully configured and executable system or application.



As the diagram shows, the Spring IoC container consumes a form of ***configuration metadata***; we can confiure metadata in following ways.

- **XML based Configuration**: Configuration metadata is traditionally supplied in a simple and intuitive XML format.
- **Annotation-based configuration**: Spring 2.5 introduced support for annotation-based configuration metadata.
- **Java-based configuration:** from Spring 3.0 onwards, you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the @Configuration, @Bean, @Import and @DependsOn annotations.

Spring – HelloWorld Example

```
public class Student {
        private String name;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public void getData() {
                System.out.println("Hello, " + name);
        }
}
```

```
//SpConfig.xml
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

        <bean id="hello" class="core.Student">
                <property name="name" value="Satya" />
        </bean>

</beans>
```

```
package core;

public class App {
        public static void main(String[] args) {

                // Instantiating a container
                ApplicationContext context = new ClassPathXmlApplicationContext("SpConfig.xml");

                Student student = (Student) context.getBean("hello");

                student.getData();
        }
}
Hello, Satya
```

Spring – Dependency Injection

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing.

In Spring framework, we can perform Dependency Injection in two ways

1.Setter Injection: we can perform DI using **setter methods** with following Datatypes

- Primitive Types
- Object Types
- Collection Types

2. Constructor Injection: we can perform DI using **Constructors** with following Datatypes

- Primitive Types
- Object Types
- Collection Types

Setter Injection with Primitive Types

```
package core;

public class Student {

        private int sno;
        private String name;
        private String address;
```

```
	public int getSno() {
		return sno;
	}

	public void setSno(int sno) {
		this.sno = sno;
	}

	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;
	}

	@Override
	public String toString() {
		return "Student [sno=" + sno + ", name=" + name + ", address=" + address + "]";
	}

}
```

```
//SpDI.xml
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="st" class="core.Student">
		<property name="sno" value="100"></property>
		<property name="name" value="Satya" />
		<property name="address" value="HYDERABAD"></property>
	</bean>

</beans>
```

```
package core;

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

public class App {
	public static void main(String[] args) {

		// Instantiating a container
		ApplicationContext context = new ClassPathXmlApplicationContext("SpDI.xml");

		Student student = (Student) context.getBean("st");
		System.out.println(student);
	}
}
Student [sno=100, name=Satya, address=HYDERABAD]
```

Setter Injection with Object Types

If our class is depending on other class object, then dependency is in the form of object

- If one spring bean is depending on another spring bean class for performing some logic, this process of dependency is called **Object dependency.**
- If object dependency is there then in spring framework, the **spring IOC container is responsible for creating that required object** and injecting into the dependent class
- For **xml**, we have 2 ways to inform to the spring container about this object dependency
    - Using **<ref />** element
    - Using Inner beans

1.Using <ref/> Tag

we can write **any number of spring configuration xmls** for the spring application. Our collaborator bean may be in **same xml or other xml** so spring has given these 3 options(local/parent/bean).

```
<ref local/parent/bean=”id of collaborator bean”>
```

1.<ref local=”id value” />

If we use the local attribute in the <ref /> element, then the spring IOC container will verify for the collaborator bean with in **same container (same xml).**

```
<beans>

  <bean id="id1">
    <property name="sb" class="Student">
      <ref local="id2" />
    </property>
  </bean>

  <bean id="id2" class="Address">

</beans>
```

2.<ref parent=”id value” />

If we use the **parenet** attribute in the <ref /> element, then the spring IOC container will verify for the collaborator bean with in **other container (other xml)**

```
//SpConfig1.xml
<beans>
        <bean id="id1">
                 <property name="sb" class="Student">
                 <ref parent="id2" />
                </property>
         </bean>
</beans>

//SpConfig2.xml
<beans>
  <bean id="id2" class="Address">
</beans>
```

3.<ref bean="id value" />

If we give attribute as bean, then **first it will check at local xml file, then parent if its not available at local**

```
public class Student {

	private int sno;
	private String name;
	private Address address;

	public int getSno() {
		return sno;
	}

	public void setSno(int sno) {
		this.sno = sno;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public Address getAddress() {
		return address;
	}

	public void setAddress(Address address) {
		this.address = address;
	}

	@Override
	public String toString() {
		return "Student [sno=" + sno + ", name=" + name + "]";
	}

}
```

```
public class Address {
	private int hno;
	private String city;

	public int getHno() {
		return hno;
	}

	public void setHno(int hno) {
		this.hno = hno;
	}

	public String getCity() {
		return city;
	}

	public void setCity(String city) {
		this.city = city;
	}

	@Override
	public String toString() {
		return "Address [hno=" + hno + ", city=" + city + "]";
	}

}
```

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="st" class="core.Student">
		<property name="sno" value="100"></property>
		<property name="name" value="Satya" />
		<property name="address">
			<ref bean="addr" />
		</property>
	</bean>

	<bean id="addr" class="core.Address">
		<property name="hno" value="200"></property>
		<property name="city" value="HYDERABAD"></property>
	</bean>

</beans>
```

```java
package core;

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

public class App {
	public static void main(String[] args) {

		// Instantiating a container
		ApplicationContext context = new ClassPathXmlApplicationContext("SpDI.xml");

		Student student = (Student) context.getBean("st");
		System.out.println(student);

		System.out.println(student.getAddress().toString());
	}
}
```

```
Student [sno=100, name=Satya]
Address [hno=200, city=HYDERABAD]
```

2.Using Inner Bean

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="st" class="core.Student">
		<property name="sno" value="100"></property>
		<property name="name" value="Satya" />

		<!-- INNER BEAN -->
		<property name="address">
			<bean id="addr" class="core.Address">
				<property name="hno" value="200"></property>
				<property name="city" value="HYDERABAD"></property>
			</bean>
		</property>
		<!-- INNER BEAN -->

	</bean>
```

Setter Injection with Collection Types

In Spring bean class, we can use any of the **following 4 types of collections** as dependency, along with Primitives Types and Objects Types

- **Set**
- **List**
- **Map**
- **Properties**

Spring supports only these 4 collections. if we use other than these Collections, programmer should have to take care about Dependency injection because Spring IoC doesn't know other collections.

1.<list >: List allows Duplicate Values

We use <value> in the case of primitive types

```
<property name="states">
        <list>
                <value>ANDHRA</value>
                <value>ANDHRA</value>
                <value>TELANGANA</value>
                <value>TAMILNADU</value>
        </list>
</property>
```

We use <ref> in the case of Object types

```
<bean id="ob" class="collectionsref.Country">
                <property name="countryName" value="INDIA"></property>
                <property name="states">
                        <list>
                                <ref bean="list1"/>
                                <ref bean="list2"/>
                        </list>
                </property>
</bean>


<bean id="list1" class="collectionsref.State">
                <property name="stName" value="ANDHRA"></property>
                <property name="stCapital" value="HYDERABAD"></property>
</bean>
```

2.<set >: Set Doesn't allow Duplicate Values

We use <value> in the case of primitive types

```
<property name="states">
        <set>
                <value>ANDHRA</value>//Not allows Duplicates
                <value>ANDHRA</value>
                <value>TELANGANA</value>
                <value>TAMILNADU</value>
        </set>
</property>
```

We use <ref> in the case of Object types

```
	<bean id="ob" class="collectionsref.Country">
		<property name="countryName" value="INDIA"></property>
		<property name="states">
			<set>
				<ref bean="set1"/>
				<ref bean="set2"/>
			</set>
		</property>
	</bean>


	<bean id="set1" class="collectionsref.State">
		<property name="stName" value="ANDHRA"></property>
		<property name="stCapital" value="HYDERABAD"></property>
	</bean>
```

3.<map> Map will accept data in <KEY, VALUE> pair, here <KEY> must be UNIQUE

We use <entry key=" " value=" ">in the case of primitive types

```
<map>
	<entry key="ANDHRA" value="VIJAYAWADA"></entry>
	<entry key="TELANGANA" value="HYDERABAD"></entry>
	<entry key="TAMILNADU" value="CHENNAI"></entry>
</map>
```

We use <entry key-ref=" " value-ref=" ">in the case of Object types

```
<map>
	<entry key-ref ="statesObj1" value-ref="capObj1"></entry>
	<entry key-ref ="statesObj2" value-ref ="capObj2"></entry>
</map>
```

```
public class Country {
	private String countryName;
	private List<State> states;

	public String getCountryName() {
		return countryName;
	}

	public void setCountryName(String countryName) {
		this.countryName = countryName;
	}

	public List<State> getStates() {
		return states;
	}

	public void setStates(List<State> states) {
		this.states = states;
	}
}
```

```
package core;

public class State {
	private String stName;
	private String stCapital;

	public String getStName() {
		return stName;
	}
```

```
	public void setStName(String stName) {
		this.stName = stName;
	}

	public String getStCapital() {
		return stCapital;
	}

	public void setStCapital(String stCapital) {
		this.stCapital = stCapital;
	}
}
```

```
//Sp1.xml

<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="ctr" class="core.Country">
		<property name="countryName" value="INDIA"></property>
		<property name="states">
			<list>
				<ref bean="st1"/>
				<ref bean="st2"/>
			</list>
		</property>

	</bean>


	<bean id="st1" class="core.State">
		<property name="stName" value="ANDRA" />
		<property name="stCapital" value="VIJAYAWADA" />
	</bean>

	<bean id="st2" class="core.State">
		<property name="stName" value="KARNATAKA" />
		<property name="stCapital" value="BANGLORE" />
	</bean>

</beans>
```

```
package core;

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

public class App2 {
	public static void main(String[] args) {

		// Instantiating a container
		ApplicationContext context = new ClassPathXmlApplicationContext("Sp1.xml");

		Country country = (Country) context.getBean("ctr");
		System.out.println(country.getCountryName());

		for (State s : country.getStates()) {
			System.out.println(s.getStName() + ": " + s.getStCapital());
		}
	}
}
```

```
INDIA
ANDRA: VIJAYAWADA
KARNATAKA: BANGLORE
```

Constructor Injection

In this type of injection Spring Container uses **constructor of the bean class** for assigning the dependencies. In **SpringConfig.xml**, we need to inform to the spring IOC container about constructor injection by using `<constructor -arg />`

In spring bean class**, if both constructor and setter injection applied** for same property then **constructor injection will be overridden by setter injection**, because constructor injection will happen at the object creation time, and setter after objection. so finally, **setter injected data will be there**.

1.Constructor Injection –Primitive Types

```
public class Student {
	private int sno;
	private String name;
	public Student(int sno, String name) {
		this.sno = sno;
		this.name = name;
	}
}
```

```
<!-- File : SpringConfig.xml -->
	<bean id="ob" class="Student">
		<constructor-arg name="sno" value="103"></constructor-arg>
		<constructor-arg name="name" value="Satya"></constructor-arg>
	</bean>
```

2.Constructor Injection –Object Types

```
public class Student {
	private int sno;
	private String name;
	private Address address;

	public Student(int sno, String name, Address address) {
		this.sno = sno;
		this.name = name;
		this.address = address;
	}
}
```

```
public class Address {
	private int hno;
	private String city;

	public Address(int hno, String city) {
		this.hno = hno;
		this.city = city;
	}
}
```

```
<beans>
	<bean id="st" class="obj.Student">
		<constructor-arg name="sno" value="101"></constructor-arg>
		<constructor-arg name="name" value="Satya Kaveti"></constructor-arg>
		<constructor-arg name="address">
			<ref bean="adr" />
		</constructor-arg>
```

```
        </bean>
        <bean id="adr" class="obj.Address">
                <constructor-arg name="hno" value="305"></constructor-arg>
                <constructor-arg name="city" value="HYDERABAD"></constructor-arg>
        </bean>
</beans>
```

3. Constructor Injection –Collection Types

```
public class Country {
        private String countryName;
        private List<State> states;
        public Country(String countryName, List<State> states) {
                super();
                this.countryName = countryName;
                this.states = states;
        }
}
```

```
public class State {
        private String stName;
        private String stCapital;
        public State(String stName, String stCapital) {
                super();
                this.stName = stName;
                this.stCapital = stCapital;
        }
}
```

```
<bean id="ob" class="collectionsref.Country">
                <constructor-arg name="countryName" value="INDIA"></constructor-arg>
                <constructor-arg name="states">
                        <list>
                                <ref bean="list1"/>
                                <ref bean="list2"/>
                        </list>
                </constructor-arg>
</bean>

<bean id="list1" class="collectionsref.State">
                <constructor-arg name="stName" value="ANDHRA"></constructor-arg>
                <constructor-arg name="stCapital" value="HYDERABAD"></constructor-arg>
</bean>
```

Example

```
public class Country {
        private String countryName;
        private List<State> states;

        public Country(String countryName, List<State> states) {
                super();
                this.countryName = countryName;
                this.states = states;
        }

        public void getCountry() {
                System.out.println("Country Name : " + this.countryName);
                List<State> states = this.states;
                Iterator<State> itr = states.iterator();
                while (itr.hasNext()) {
                        State s = (State) itr.next();
                        s.getState();
                }
        }
}
```

```java
package core;

public class State {
	private String stName;
	private String stCapital;

	public State(String stName, String stCapital) {
		this.stName = stName;
		this.stCapital = stCapital;
	}
	public void getState() {
		System.out.println(this.stName + ", " + this.stCapital);
	}
}
```

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="ob" class="core.Country">
		<constructor-arg value="INDIA"></constructor-arg>
		<constructor-arg>
			<list>
				<ref bean="list1" />
				<ref bean="list2" />
			</list>
		</constructor-arg>
	</bean>

	<bean id="list1" class="core.State">
		<constructor-arg value="ANDHRA"></constructor-arg>
		<constructor-arg value="HYDERABAD"></constructor-arg>
	</bean>

	<bean id="list2" class="core.State">
		<constructor-arg value="TAMILNADU"></constructor-arg>
		<constructor-arg value="CHENNAI"></constructor-arg>
	</bean>

</beans>
```

```java
public class App2 {
	public static void main(String[] args) {

		Resource resource = new ClassPathResource("Sp1.xml");
		BeanFactory factory = new XmlBeanFactory(resource);

		Object ob = factory.getBean("ob");
		Country c = (Country) ob;
		c.getCountry();
	}
}
```

```
Country Name : INDIA
ANDHRA, HYDERABAD
TAMILNADU, CHENNAI
```

Setter Injection	Constructor Injection
1.Partial injection possible: if we have 3 dependencies like int, string, long, then its not necessary to inject all values if we use setter injection. **If you are not inject it will takes default values for those primitives**	**1.Partial injection NOT possible:** for calling constructor we must pass all the arguments, otherwise we will get Error.

2. Setter Injection will overrides the constructor injection value, provided if we write setter and constructor injection for the same property .	**2.** Constructor injection cannot overrides the setter injected values
3. If we have more dependencies for example 15 to 20 are there in our bean class then, in this case setter injection is not recommended as we need to write almost 20 setters right, bean length will increase.	**3.** In this case, Constructor injection is highly recommended, as we can inject all the dependencies with in 3 to 4 lines .
4. Setter injection makes bean class object as mutable i.e We can change	**4.** Constructor injection makes bean class object as immutable.i.e We cannot change

Spring – Aurowire

In previous Examples, for Dependency injection we wrote the bean properties explicitly into SpringConfig.xml file.

By using Autowiring **we no need to write the bean properties explicitly into SpringConfig.xml,** because **Spring Container will take care about injecting the dependencies.**

- **By default,**autowiring is **disabled** in spring framework.
- **Autowiring** supports only **Object types**, Not Primitive, Collection types

In Spring, 5 Auto-wiring modes are supported.

- **byName [ID comparison]**
- **byType [CLASS TYPE comparison]**
- **Constructor**
- **autoDetect**
- **no**

To activate Autowire in our application we need to configure autowire attribute in <bean> tag, with any one of above 5 modes. The syntax will be like below

```
<bean id=”id” class=”class” autowire="byName/byType/constructor/autoDetect/no">
```

1.by Name

- In this mode, spring framework will try to find out a bean in the SpringConfig.xml file, whose **bean id** is matching with the **property name** to be wired.
- If a bean found with id as property name, then that class object will be injected into that property by **calling setter injection**
- If no id is found then that property remains un-wired, but **never throws any exception**.

```
package core;

public class Student {

	private int sno;
	private String name;
	private Address address;

	public int getSno() {
		return sno;
	}

	public void setSno(int sno) {
		this.sno = sno;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public Address getAddress() {
		return address;
	}

	public void setAddress(Address address) {
		this.address = address;
	}

	@Override
	public String toString() {
		return "Student [sno=" + sno + ", name=" + name + "]";
	}

}
```

```
package core;

public class Address {
	private int hno;
	private String city;

	public int getHno() {
		return hno;
	}

	public void setHno(int hno) {
		this.hno = hno;
	}

	public String getCity() {
```

```
		return city;
	}

	public void setCity(String city) {
		this.city = city;
	}

	@Override
	public String toString() {
		return "Address [hno=" + hno + ", city=" + city + "]";
	}

}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="st" class="core.Student" autowire="byName">
		<property name="sno" value="100"></property>
		<property name="name" value="Satya" />

		<!-- This is Not Required
		<property name="address">
			<ref bean="address"/>
		</property>
		 -->
	</bean>

	<bean id="address" class="core.Address">
		<property name="hno" value="200"></property>
		<property name="city" value="HYDERABAD"></property>
	</bean>


</beans>
```

```
package core;

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

public class App {
	public static void main(String[] args) {

		// Instantiating a container
		ApplicationContext context = new ClassPathXmlApplicationContext("SpDI.xml");

		Student student = (Student) context.getBean("st");
		System.out.println(student);

		System.out.println(student.getAddress().toString());
	}
}
Student [sno=100, name=Satya]
Address [hno=200, city=HYDERABAD]
```

In above example spring container compares the `<bean id="address">` with bean property `private Address address;`

2.byType

In '**byType**'' mode, if data type of a bean in SpringConfig.xml is matched with data type of the Bean Property in bean class, it will autowire the properties **using Setter Injection.**

```
<beans>
	<bean>
		<bean id="student" class="core.Student" autowire="byType">
			<property name="sno" value="101"></property>
			<property name="name" value="Satya Kaveti"></property>
		</bean>

		<bean id="address" class="core.Address">
			<property name="hno" value="322"></property>
			<property name="city" value="HYDERABAD"></property>
		</bean>


		<bean id="address1" class="core.Address">
			<property name="hno" value="322"></property>
			<property name="city" value="HYDERABAD"></property>
		</bean>
</beans>
```

In above you have multiple bean of same type, Container will confuse which bean should inject & throws **NoSuchBeanDefinitionException:**

```
by: org.springframework.beans.factory.NoSuchBeanDefinitionException: No unique bean of type [core.Address]
is defined: expected single matching bean but found 2: [address, address1]
```

To fix above problem, you need **@Qualifier** to tell Spring about which bean should autowired.

```
public class Student {

	private int sno;
	private String name;
	@Qualifier("address")
	private Address address;
}
```

3. Constructor

- Autowiring by constructor **is similar to byType**, but here **it will use Constructor for injection** instead of Setter methods.
- In this case we have to write the Constructor for Bean Property, but not Setter methods. That means we have write Constructor for address property instead of setAddress() method.
- In there are multiple constructors **like one-arg, two-arg, three-arg**, it will take **three-arg** constructor for injecting properties. i.e. **Max-arg Param constructor will do the job**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- File : SpringConfig.xml -->
<beans>

	<bean id="student" class="constructor.Student" autowire="constructor">
		<property name="sno" value="101"></property>
		<property name="name" value="Satya Kaveti"></property>
	</bean>
```

```
        <bean id="address" class="constructor.Address">
                <property name="hno" value="322"></property>
                <property name="city" value="HYDERABAD"></property>
        </bean>
</beans>
```

4.autodetect

- **autowire="autodetect"** first will works as **constructor autowire** if not, then works **byType** as Autowiring.
- It is deprecated since Spring 3.

5.no

autowire="no" is the default autowiring mode. It means no autowiring by default

Spring – Autowire using Annotations

Starting with Spring 2.5, the framework introduced a new style of Dependency Injection driven by *@Autowired* Annotations. This annotation allows Spring to resolve and inject collaborating beans into your bean.

To enable Annotation based autowiring we need to place below line in SpringConfig.xml

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

Once annotation injection is enabled, autowiring can be used on properties, setters, and constructors.

1. *@Autowired* on Properties

```
public class Student {
        private int sno;
        private String name;

        @Autowired
        private Address address;

}
```

2. ***@Autowired* on Setters**

```
public class Student {
        private int sno;
        private String name;

        private Address address;

        @Autowired
        public void setAddress(Address address) {
                this.address = address;
        }
}
```

3. *@Autowired* on Constructors

```
public class Student {
	private int sno;
	private String name;

	private Address address;

	@Autowired
	public Student(Address address) {
		this.address = address;
	}
}
```

4.@Autowired and Optional Dependencies

Spring expects *@Autowired* dependencies to be available when the dependent bean is being constructed. If the framework cannot resolve a bean for wiring, it will throw **NoSuchBeanDefinitionException.** To avoid this we have to use **(required=false)**

```
public class Student {
	private int sno;
	private String name;

	@Autowired(required = false)
	private Address address;
}
```

By default, the @Autowired annotation implies the dependency is required similar to @Required annotation, however, you can turn off the default behavior by using **(required=false)** option with @Autowired.

5.Autowiring by *@Qualifier*

By default, Spring resolves *@Autowired* entries by type. If more than one beans of the same type are available in the container, the framework will throw a fatal exception indicating that more than one bean is available for autowiring.

To avoid this error we have to use @Qualifier

```
<beans>
	<bean>
		<bean id="student" class="core.Student" autowire="byType">
			<property name="sno" value="101"></property>
			<property name="name" value="Satya Kaveti"></property>
		</bean>

		<bean id="address" class="core.Address">
			<property name="hno" value="322"></property>
			<property name="city" value="HYDERABAD"></property>
		</bean>


		<bean id="address1" class="core.Address">
			<property name="hno" value="322"></property>
			<property name="city" value="HYDERABAD"></property>
		</bean>
</beans>
```

In above you have multiple bean of same type, Container will confuse which bean should inject & throws **NoSuchBeanDefinitionException:**

```
by: org.springframework.beans.factory.NoSuchBeanDefinitionException: No unique bean of type [core.Address]
is defined: expected single matching bean but found 2: [address, address1]
```

To fix above problem, you need **@Qualifier** to tell Spring about which bean should autowired.

```
public class Student {

	private int sno;
	private String name;
	@Qualifier("address")
	private Address address;
}
```

Spring – Spring Bean Internal Working

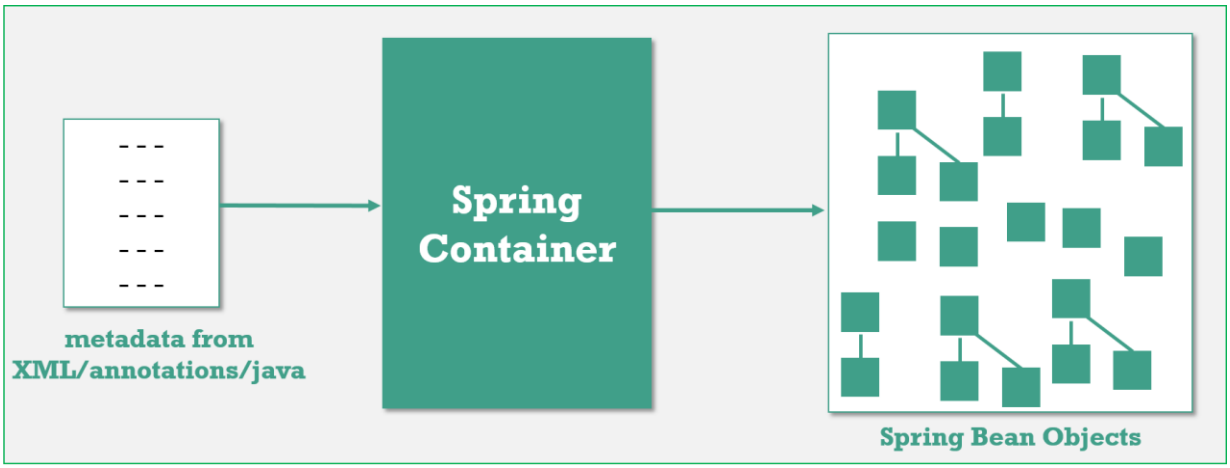
Spring IoC container receives metadata from either an **XML file, Java annotations, or Java code**.

By reading the configuration metadata container gets its instructions, which POJO class objects to instantiate, configure, and assemble.

The Objects which are created through this process called **Spring Beans**.

The responsibilities of IoC container are:

- **Instantiating the bean**
- **Wiring the beans together**
- **Configuring the beans**
- **Managing the bean's entire life-cycle**



If you look at the picture above, you can see that there is a lot of squares in there.

That's how the spring container would look like if you graphed it all out. The little squares are the **Spring beans** and you can see **their references**. Some of them are standalone, some of them are referencing other beans. Just how they wired up and how it makes all these objects that we're using altogether.

Spring <bean> configuration

A Spring IoC container manages one or more *beans*. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML **<bean/>** definitions.

bean definitions are represented as **BeanDefinition** objects, which contain following metadata:

```
<beans>

        <bean id="student" name="student" class="core.Student" scope="prototype"
                autowire="byName" dependency-check="simple" lazy-init="true/false" init-method="
                initIt " destroy-method="cleanUp">

        </bean>

        <import resource="connection/Spring-Connection.xml" />
        <import resource="moduleA/Spring-ModuleA.xml" />

</beans>
```

1.Id

A bean will have only one Unique id, **special charaters are not allowd.**

2.name

A single bean can have multiple names(aliases) & allowes Special Charaters, However, the names must still unique, otherwise **BeanDefinitionParsingException** – Bean name 'kingBean' is already used in this file

```
<bean id="foo" name = "myFoo,kingBean,notBar" class="com.Foo">
</bean>
```

3.class

This attribute is mandatory and specify the bean class to be used to create the bean. You should specify fully qualified class name. Include package name.

4.scope

5 types of bean scopes supported:

1. **singleton(default)** – Return a single bean instance per Spring IoC container

2. **prototype** – Return a new bean instance each time when requested
3. **request** – each HTTP request has its own instance of a bean created
4. **session** – Scopes a single bean definition to the lifecycle of an HTTP Session
5. **application** – Scopes a single bean definition to the lifecycle of a ServletContext

The `request`, `session`, and `application` scopes are *only* available if you use a web-aware Spring `ApplicationContext` implementation (such as `XmlWebApplicationContext`). If you use these scopes with regular Spring IoC containers such as the `ClassPathXmlApplicationContext`, you get an `IllegalStateException` complaining about an unknown bean scope.

By XML

```
<bean id="student" class="core.Student" scope="singleton/prototype/request/session/application">
        ---
</bean>
```

You can also use annotation to define your bean scope.

```
@Service
@Scope("prototype")
public class CustomerService
{
        String message;

        public String getMessage() {
                return message;
        }

        public void setMessage(String message) {
                this.message = message;
        }
}
```

Enable auto component scanning

```
<beans>
      <context:component-scan base-package="com.smlcodes.customer" />
</beans>
```

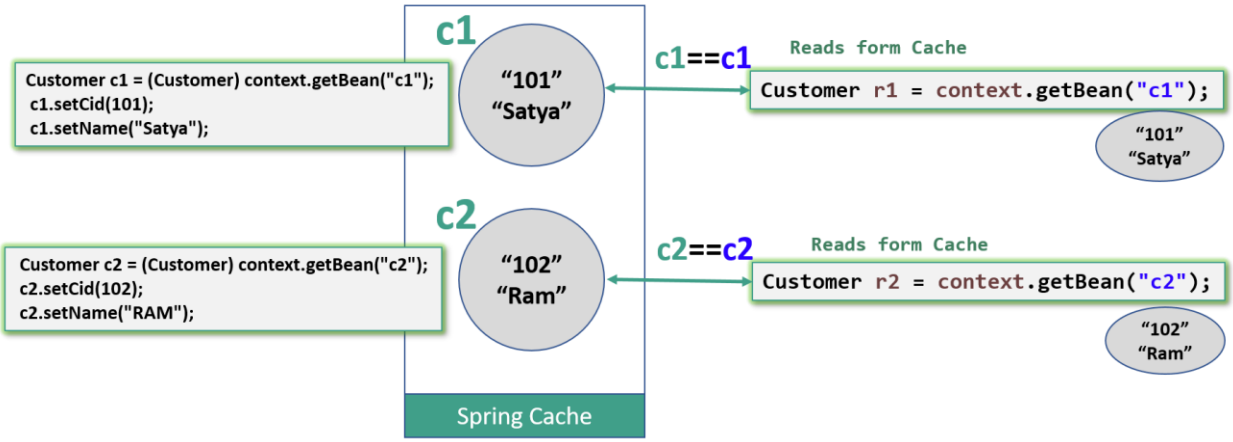
SingleTon Vs Prototype

Singleton

when you define a bean definition and it is scoped as a **singleton**, the Spring IoC container creates **exactly one instance with that particular ID** of the object defined by that bean definition.

This **single instance is stored in a cache** of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The singleton scope is the default scope in Spring.




```java
package core;

public class Customer {

	private int cid;
	private String name;

	public int getCid() {
		return cid;
	}

	public void setCid(int cid) {
		this.cid = cid;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

}
```

```xml
//Default is 'SingleTon'
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="c1" class="core.Customer">
	</bean>

	<bean id="c2" class="core.Customer">
	</bean>


</beans>
```

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

		// Instantiating a container
		ApplicationContext context = new ClassPathXmlApplicationContext("scope.xml");

	// ********** Setting Data
		Customer c1 = (Customer) context.getBean("c1");
```

```
                c1.setCid(101);
                c1.setName("Satya");

                Customer c2 = (Customer) context.getBean("c2");
                c2.setCid(102);
                c2.setName("RAM");


        //********* retrieve it again
                Customer r1 = (Customer) context.getBean("c1");
                System.out.println("R1 ==>" +r1.getCid()+" : "+r1.getName());


                Customer r2 = (Customer) context.getBean("c2");
                System.out.println("R2 ==>" +r2.getCid()+" : "+r2.getName());

        }
}
```

```
R1 ==>101 : Satya
R2 ==>102 : RAM
```

Since the bean 'Customer' is in singleton scope, the second retrieval by '**R1, R2** will display the same data set by **'C1, C2'** also, even it's retrieve by a new getBean() method.

In singleton, only a single instance per Spring IoC container, no matter how many time you retrieve it with getBean(), it will always return the same instance

Prototype

The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made.

If you change **scope="*prototype*"** the Spring.Xml , it will create new bean every time its called.

```
<beans>
        <bean id="c1" class="core.Customer" scope="prototype">
        </bean>

        <bean id="c2" class="core.Customer" scope="prototype">
        </bean>
</beans>
```

You will get following output.

```
R1 ==>0 : null
R2 ==>0 : null
```

In prototype scope, you will have a new instance for each `getBean()` method called.

6.dependency-check

In Spring, you can use dependency checking feature to make sure the required properties have been set or injected.

4 dependency checking modes are supported:

- **none (default)** – No dependency checking.
- **simple** – If any properties of primitive type (int, long, double…) and collection types (map, list..) have not been set, **UnsatisfiedDependencyException** will be thrown.
- **objects** – If any properties of object type have not been set, **UnsatisfiedDependencyException** will be thrown.
- **all** – If any properties of any type have not been set, an **UnsatisfiedDependencyException** will be thrown

Explicitly define the dependency checking mode for every bean class is tedious and error prone, you can set a default-dependency-check attribute in the <beans> root element to force the entire beans declared within <beans> root element to apply this rule. However, this root default mode will be overridden by a bean's own mode if specified.

In XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        default-dependency-check="all">

       <bean id="CustomerBean" class="com.smlcodes.Customer" dependency-check="simple/objects/all">
                <property name="person" ref="PersonBean" />
                <property name="action" value="buy" />
       </bean>

</beans>
```

Using Annotations

Spring's dependency checking in bean configuration file is used to make sure all properties of a certain types (primitive, collection or object) have been set. In most scenarios, you just need to make sure a particular property has been set, but not all properties.

For this case, you need **@Required** annotation, see following example :

A Customer object, apply @Required in setPerson() method to make sure the person property has been set.

```
public class Customer
{
        private Person person;
        private int type;
        private String action;

        public Person getPerson() {
                return person;
        }
        @Required
        public void setPerson(Person person) {
                this.person = person;
        }
}
```

register an **RequiredAnnotationBeanPostProcessor** to aware of the **@Required** annotation in bean configuration file.

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

```
        ...
</beans>
```

7.Import

Using Java : You may load multiple Spring bean configuration files in the code :

```
ApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"Spring-Common.xml",
              "Spring-Connection.xml","Spring-ModuleA.xml"});
```

Using xml : SpringAll.xml

```
        <import resource="common/Spring-Common.xml"/>
        <import resource="connection/Spring-Connection.xml"/>
        <import resource="moduleA/Spring-ModuleA.xml"/>
```

Now you can load a single xml file like this :

```
ApplicationContext context = new ClassPathXmlApplicationContext(SpringAll.xml);
```

8.Lazy-init

By default, Spring "application context" eagerly creates and initializes all beans during application startup itself. It helps in detecting the bean configuration issues at early stage, in most of the cases.

But sometimes, you may need to mark some or all beans to be lazy initialized, means Beans should initailze when they are required, by using **lazy-init** attribute.

```
public class A {
        public A(){
                System.out.println("-- A initialized");
        }
}
```

```
public class B {
        public B() {
                System.out.println("-- B initialized");
        }
}
```

```
<beans>
         <bean id="a" class="core.A" lazy-init="default" />
         <bean id="b" class="core.B" lazy-init="true" />
</beans>
```

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

                System.out.println("**** Eager Inits START ****");
                ApplicationContext context = new ClassPathXmlApplicationContext("Spring.xml");
                System.out.println("**** Eager Inits End ****");

                System.out.println("Lazy Init ...Intializes Only when calling getBean()");
                context.getBean("b");
```

```
        }
}
```

```
**** Eager Inits START ****
-- A initialized
**** Eager Inits End ****
Lazy Init ...Intializes Only when calling getBean()
-- B initialized
```

Using XML

1.Lazy load specific beans only

```
<beans>
        <bean id="e" class="EmployeeManagerImpl"  lazy-init="true"/>
<beans>
```

2.Lazy load all beans

```
<beans default-lazy-init="true">
        <bean id="e" class="EmployeeManagerImpl" />
<beans
```

Using Annotations

1.Lazy load specific beans only

```
@Configuration
public class AppConfig {

        @Lazy
        @Bean
        public EmployeeManager employeeManager() {
                return new EmployeeManagerImpl();
        }
}
```

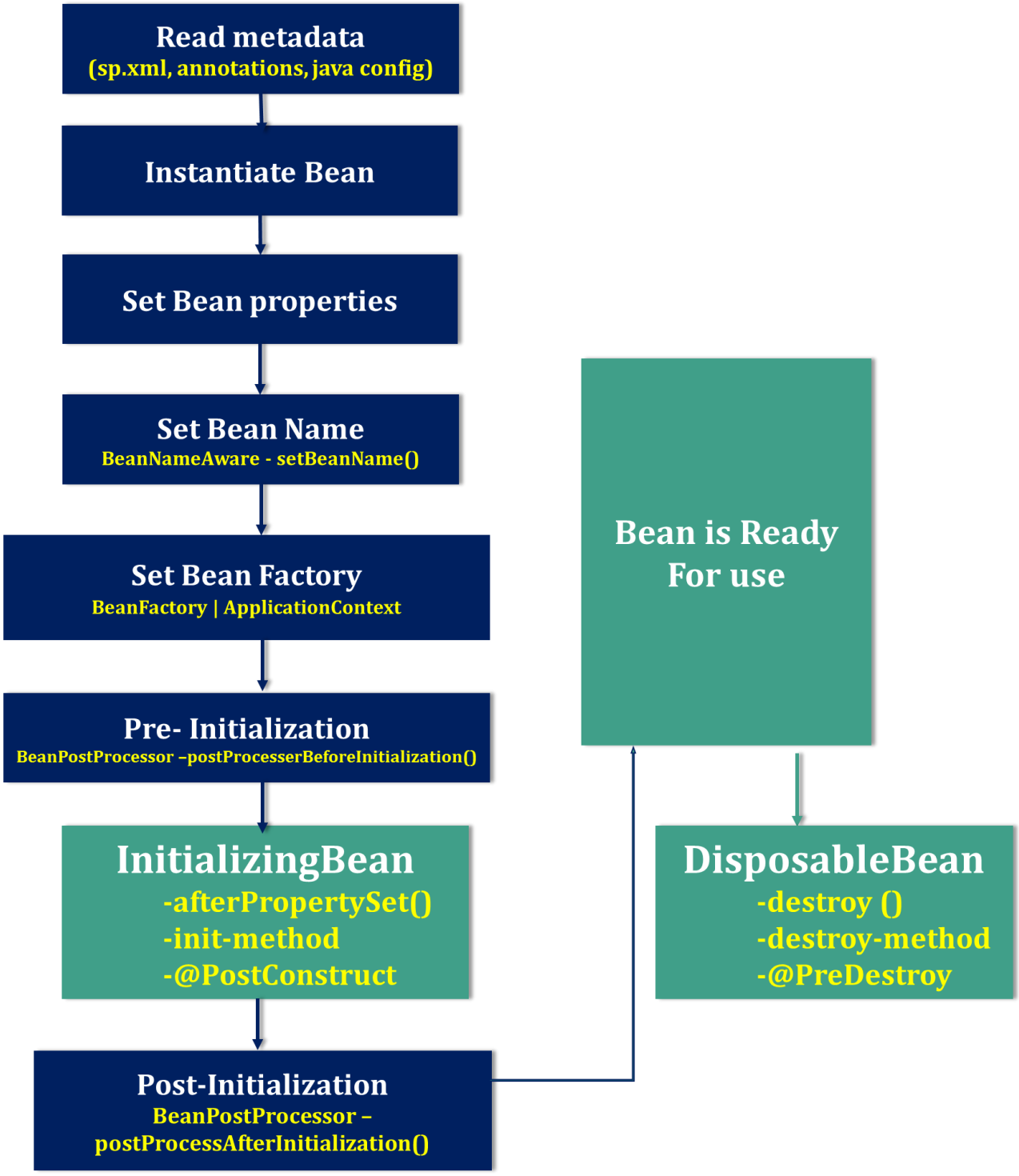
2.Lazy load all beans

```
@Lazy
@Configuration
public class AppConfig {

        @Bean
        public EmployeeManager employeeManager() {
                return new EmployeeManagerImpl();
        }

}
```

init-method, destroy-method

to understand these, you must know Bean Lify Cycle, Let's Start!!

Spring Bean Life Cycle



1.Read metadata

Spring container reads metadata from **SpringConfiguration.xml (or annotations/java config)** file and looks for the **<bean>** definitions.

2.Instantiate

Spring **instantiate** the bean by calling no argument default constructor of that class, If there is **only parameterized constructor** in the class, then bean must be defined in **spring.xml** file with **constructor injection** otherwise it will throw **BeanCreationException**.

3.Inject Bean properties

Once instantiate completed, **Sprin injects the values and references into the bean's properties**.

4.Set Bean Name

If the bean implements **BeanNameAware** interface, Spring executes **setBeanName()** method by passing Bean Id(bean id="ob"). By this method Spring container sets the **bean name.**

5.Set Bean Factory

- If the bean implements **BeanFactoryAware** interface, Spring executes **setBeanFactory()** method by passing current **BeanFactory** reference which is used in our Appplciation.

- If the bean implements **ApplicationContextAware** interface, Spring executes **setApplicationContext()** method by passing current **ApplicationContext** reference which is used in our Appplciation.

6.Pre- Initialization

Apply this **BeanPostProcessor** to the given new bean instance **before any bean initialization callbacks** (like InitializingBean's afterPropertiesSet or a custom init-method) by using **postProcesserBeforeInitialization()** method. . The bean will already be populated with property values. Note it says that "*The bean will already be populated with property values*"

7. Initialize beans

- If the bean implements **IntializingBean**,its **afterPropertySet()**method is called.
- If the bean has custom **init-method**, then specified initialization method is called.
- If we are using annotations, use **@PostConstruct** on the Top of the method

8.Post-Initialization

Apply this **BeanPostProcessor** to the given new bean instance after any bean initialization callbacks (after InitializingBean's, afterPropertiesSet, custom init-method) by **postProcessAfterInitialization().**

9.Ready to Use

Now the bean is ready to be used by the application.

10. DisposableBean

- If the bean implements **DisposableBean**, the Spring IoC container will call the **destroy()** method .
- If a custom **destroy-method** is defined, the container calls the specified method.
- If we are using annotations, use **@PreDestroy** on the Top of the method

Spring Bean Lifecycle Example

```
package lifycycle;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class Student implements BeanNameAware, BeanPostProcessor, InitializingBean, DisposableBean {

	public void setBeanName(String beanname) {
		System.out.println("setBeanName : " + beanname);

	}

	public Object postProcessBeforeInitialization(Object arg0, String arg1) throws BeansException {
		System.out.println("BeanPostProcessor : postProcessBeforeInitialization ");
		return null;
	}

	public Student() {
		System.out.println("Student Contrscutor...");
	}

	public Object postProcessAfterInitialization(Object arg0, String arg1) throws BeansException {
		System.out.println("BeanPostProcessor : postProcessAfterInitialization ");
		return null;
	}

	private int sno;
	private String name;
	private Address address;

	public int getSno() {
		return sno;
	}

	public void setSno(int sno) {
		System.out.println("\t SNO Property Set");
		this.sno = sno;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		System.out.println("\t NAME Property Set");
		this.name = name;
	}

	public Address getAddress() {
		return address;
	}
```

```
	public void setAddress(Address address) {
		System.out.println("\t ADDRESS OBJECT Property Set");
		this.address = address;

	}

	public void afterPropertiesSet() throws Exception {
		System.out.println("InitializingBean : afterPropertiesSet");
	}

	public void destroy() throws Exception {
		System.out.println("DisposableBean : destroy");
	}

}
```

```
package lifycycle;

public class Address {
	private String city;

	public String getCity() {
		return city;
	}

	public void setCity(String city) {
		System.out.println("\t \t CITY Property Set");
		this.city = city;
	}

}
```

SpringConfig.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="st" class="lifycycle.Student">
		<property name="sno" value="101" />
		<property name="name" value="Satya" />
		<property name="address">
			<ref bean="addr" />
		</property>
	</bean>

	<bean id="addr" class="lifycycle.Address">
		<property name="city" value="HYDERABAD" />
	</bean>

</beans>
```

```
public class App {

	public static void main(String[] args) {

		ApplicationContext context = new ClassPathXmlApplicationContext("SpringConfig.xml");
		Student s = (Student) context.getBean("st");
		System.out.println(s.getSno());
		System.out.println(s.getName());
		System.out.println(s.getAddress().getCity());
	}
}
```

```
Student Contrscutor...
                CITY Property Set
Jan 03, 2019 12:43:46 PM
org.springframework.context.support.AbstractApplicationContext$BeanPostProcessorChecker
postProcessAfterInitialization
INFO: Bean 'addr' of type [class lifycycle.Address] is not eligible for getting processed by all
BeanPostProcessors (for example: not eligible for auto-proxying)
         SNO Property Set
         NAME Property Set
         ADDRESS OBJECT Property Set
setBeanName : st
InitializingBean : afterPropertiesSet
Jan 03, 2019 12:43:46 PM org.springframework.beans.factory.support.DefaultListableBeanFactory
preInstantiateSingletons
INFO: Pre-instantiating singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@2a098129: defining beans [st,addr];
root of factory hierarchy
101
Satya
HYDERABAD
```

Configuring metadata

Spring IoC container consumes a form of *configuration metadata*; this configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application.

We can configurte the metadata in following ways

1.XML Based

You have already seen XML based configuration metadata. The following example shows the basic structure of XML-based configuration metadata:

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

        <bean id="..." class="...">
                <!-- collaborators and configuration for this bean go here -->
        </bean>

        <bean id="..." class="...">
                <!-- collaborators and configuration for this bean go here -->
        </bean>

        <!-- more bean definitions go here -->

</beans>
```

2.Annotation based

Spring 2.5 introduced support for annotation-based configuration metadata.

For using annotation based cpnfiguration we need to place following line of code in our SpringConfig.xml

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

<context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined. This means that, if you put <context:annotation-config/> in a WebApplicationContext for a DispatcherServlet, it only checks for @Autowired beans in your controllers, and not your services.

XML vs Annotation Configuration

- XML excels at wiring up components without touching their source code or recompiling them. annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.
- Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches.

Annotations

1.@Required

- The @Required annotation applies to bean property setter methods.
- This annotation indicates that the annotated bean property must be populated at configuration time, otherwise it will throws NullPointerException in case of Object type & it will take default values in case of Primitive Types
- It should be placed at the top of Setter methods, if we place other palces it shows: **@Required is disallowed for this location**

2. @Autowired

- You can apply the @Autowired annotation to constructors:
- you can also apply the @Autowired annotation to setter methods & varibles

3. @Inject

- Instead of @Autowired, we can use @Inject
- As with @Autowired, it is possible to use @Inject at the field level, method level and constructor-argument level.

- The @Inject annotation also serves the same purpose, but the main difference between them is that @Inject is a **standard annotation for dependency injection(JSR-330)** and @Autowired **is spring specific**

Spring	JSR-330	
@Autowired	@Inject	Has no 'required' attribute
@Component	@Named	
@Scope	@Scope	Only for meta-annotations and injection points
@Scope	@Singleton	Default scope is line 'prototype'
@Qualifier	@Named	
@Value	X	
@Required	X	
@Lazy	X	

- You can potentially avoid that development effort by using standard annotations specified by JSR-330 e.g. **@Inject, @Named, @Qualifier, @Scope and @Singleton**.

- A bean declared to be auto-wired using @Inject will work in both **Google Guice and Spring framework**, and potentially any other DI container which supports JSR-330 annotations.

4. @Primary

- Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process.

- @Primary indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency.

- We can also use @Qualifier annotation for the same purpose. The difference is in Qualifier we can pass paramaters.

```
public class MovieConfiguration {
        @Bean
        @Primary
        public MovieCatalog firstMovieCatalog() { ... }

        @Bean
        public MovieCatalog secondMovieCatalog() { ... }
        // ...
}

public class MovieRecommender {

        @Autowired
        @Qualifier("main")
        private MovieCatalog movieCatalog;
        // ...
}
```

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(@Qualifier("main")MovieCatalog movieCatalog,
                    CustomerPreferenceDao customerPreferenceDao) {
            this.movieCatalog = movieCatalog;
            this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}

<beans>

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>
        <!-- inject any dependencies required by this bean -->
</bean>

<bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>
        <!-- inject any dependencies required by this bean -->
</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

5.@Resource

- We can use @Resource annotation on fields or bean property setter methods.
- **@Resource** takes a name attribute, and by default Spring interprets that value as the bean name to be injected.
- If no name is specified explicitly, the default name is derived from the field name or setter method

```
public class SimpleMovieLister {

        private MovieFinder movieFinder;

        @Resource(name="myMovieFinder")
        public void setMovieFinder(MovieFinder movieFinder) {
                this.movieFinder = movieFinder;
        }

}
```

Spring component model elements vs. JSR-330 variants

Spring	javax.inject.*	javax.inject restrictions / comments
@Autowired	@Inject	@Inject has no 'required' attribute; can be used with Java 8's Optional instead.
@Component	@Named / @ManagedBean	JSR-330 does not provide a composable model, just a way to identify named components.
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring's prototype. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring's @Scope annotation. javax.inject also provides a @Scopeannotation. Nevertheless, this one is only intended to be used for creating your own annotations.
@Qualifier	@Qualifier / @Named	javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring's @Qualifier with a value) can be associated through javax.inject.Named.
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	javax.inject.Provider is a direct alternative to Spring's ObjectFactory, just with a shorter get() method name. It can also be used in combination with Spring's @Autowiredor with non-annotated constructors and setter methods.

3.Java based

Java based configuration is introduced in Spring 3.0 onwards. we have mainly @Bean ,@Configuration Annotations

```
Student.java
public class Student {
	private int sno;
	private String name;
	public int getSno() {
		return sno;
	}
	public void setSno(int sno) {
		this.sno = sno;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

}
```

Annotate with `@Configuration` to tell Spring that this is the core Spring configuration file, and define bean via `@Bean`.

```
AppConfig.java
@Configuration
public class AppConfig {
	@Bean(name = "student")
	public Student studentBean() {
		return new Student();
	}

}
```

Load your JavaConfig class with AnnotationConfigApplicationContext.

```
App.java
public class App {
	public static void main(String[] args) {
		ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
		Student s = (Student) context.getBean("student");
	}
}
```

1.@Configuration

@Configuration to tell Spring that this is the core Spring configuration file

2.@Bean

Indicates that a method produces a bean to be managed by the Spring container. This is one of the most used and important spring annotation. @Bean annotation also can be used with parameters like name, initMethod and destroyMethod.

- name – allows you give name for bean
- init-Method – allows you to choose method which will be invoked on context register
- destroy-Method – allows you to choose method which will be invoked on context shutdown

```
@Configuration
public class AppConfig {

    @Bean(name = "comp", initMethod = "turnOn", destroyMethod = "turnOff")
    Computer computer(){
        return new Computer();
    }
}
Copy
public class Computer {

    public void turnOn(){
        System.out.println("Load operating system");
    }
    public void turnOff(){
        System.out.println("Close all programs");
    }
}
```

Spring Annotations

DI Related Annotations

1.@Autowired

We can use the *@Autowired* to **mark a dependency which Spring is going to resolve and inject**. We can use this annotation with a constructor, setter, or field injection.

Field injection:

```
class Car {
    @Autowired
    Engine engine;
}
```

Setter injection:

```
class Car {
    Engine engine;

    @Autowired
    void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

Constructor injection:

```
class Car {
    Engine engine;

    @Autowired
    Car(Engine engine) {
        this.engine = engine;
    }
}
```

2. *@Bean*

@Bean marks a factory method which instantiates a Spring bean:

```
@Bean
Engine engine() {
    return new Engine();
}
```

Spring calls these methods when a new instance of the return type is required.

The resulting bean has the same name as the factory method. If we want to name it differently, we can do so with the *name* or the *value* arguments of this annotation (the argument *value* is an alias for the argument *name*):

```
@Bean("engine")
Engine getEngine() {
    return new Engine();
}
```

Note, that all methods annotated with *@Bean* must be in *@Configuration* classes.

3. *@Qualifier*

We use *@Qualifier* along with *@Autowired* to **provide the bean id or bean name** we want to use in ambiguous situations.

```
@Autowired
@Qualifier("bike")
void setVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
}
```

4. *@Required*

@Required on setter methods to mark dependencies that we want to populate through XML:

```
@Required
void setColor(String color) {
    this.color = color;
}
```

```
<bean class="com.baeldung.annotations.Bike">
    <property name="color" value="green" />
</bean
```

Otherwise, *BeanInitializationException* will be thrown.

5. *@Value*

We can use @Value for injecting property values into beans. It's compatible with constructor, setter, and field injection.

```
Field injection:
@Value("8")
int cylinderCount

Setter injection:
@Autowired
void setCylinderCount(@Value("8") int cylinderCount) {
    this.cylinderCount = cylinderCount;
}

Constructor injection:

Engine(@Value("8") int cylinderCount) {
    this.cylinderCount = cylinderCount;
}
```

6.@Scope

We use *@Scope* to define the scope of a *@Component* class or a *@Bean* definition. It can be either *singleton, prototype, request, session, globalSession* or some custom scope.

```
@Component
@Scope("prototype")
class Engine {}
```

Stereotype Annotations

Spring provides 4 stereotype annotations: @Component, @Repository, @Service, and @Controller.

@Component This is a generic annotation and can be applied to any class of the application to make it a spring managed component(simply, the generic stereotype for any spring managed component). when the classpath is scanned by the spring's component-scan **(@ComponentScan)** feature, it will identify the classes annotated with **@Component** annotation (within the given package) and create the beans of such classes and register them in the ApplicationContext. **@Component** is a class level annotation and its purpose is to make the class as spring managed component and auto-detectable bean for classpath scanning feature

ANNOTATION	USE	DESCRIPTION
@Component	Type	Generic stereotype annotation for any Spring-managed component.
@Controller	Type	Stereotypes a component as a Spring MVC controller.

ANNOTATION	USE	DESCRIPTION
@Repository	Type	Stereotypes a component as a repository. Also indicates that SQLExceptions thrown from the component's methods should be translated into Spring DataAccessExceptions.
@Service	Type	Stereotypes a component as a service.

`@Component` (and `@Service` and `@Repository`) are used to auto-detect and auto-configure beans using classpath scanning.

- @Component is a generic stereotype for any Spring-managed component.
- @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.
- Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects.
- For example, these stereotype annotations make ideal targets for pointcuts. It is also possible that @Repository, @Service, and @Controller may carry additional semantics in future releases of the Spring Framework.
- Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice. Similarly, as stated above, @Repository is already supported as a marker for automatic exception translation in your persistence layer.

@Bean vs @Componenet

1. @Component **auto detects** and configures the beans using classpath scanning whereas @Bean **explicitly declares** a single bean, rather than letting Spring do it automatically.
2. @Component **does not decouple** the declaration of the bean from the class definition where as @Bean **decouples** the declaration of the bean from the class definition.
3. @Component is a **class level annotation** where as @Bean is a **method level annotation** and name of the method serves as the bean name.
4. @Component **need not to be used with the @Configuration** annotation where as @Bean annotation has to be **used within the class which is annotated with @Configuration**.

5. We **cannot create a bean** of a class using @Component, if the class is outside spring container whereas we **can create a bean** of a class using @Bean even if the class is present **outside the spring container**.

6. @Component has **different specializations** like @Controller, @Repository and @Service whereas @Bean has **no specializations**.

Context Configuration Annotations

1.@Profile

The Spring `@Profile` allow developers to register beans by condition. For example load a database properties file based on the application running in development, test, staging or production environment.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("development")
public @interface Production {
}
```

2.@Import annotation

Normally, you will split a large Spring XML bean files into multiple small files, group by module or category, to make things more maintainable and modular. For example,

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

        <import resource="config/customer.xml"/>
         <import resource="config/scheduler.xml"/>

</beans>
```

In Spring3 JavaConfig, the equivalent functionality is @Import.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({ CustomerConfig.class, SchedulerConfig.class })
public class AppConfig {
```

```
}
```

3. *@ImportResource*

We can **import XML configurations** with this annotation. We can specify the XML file locations with the *locations* argument, or with its alias, the *value* argument:

```
@Configuration
@ImportResource("classpath:/annotations.xml")
class VehicleFactoryConfig {}
```

4.@PropertySource

With this annotation, we can **define property files for application settings**.
@PropertySource leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple times:

```
@Configuration
@PropertySource("classpath:/annotations.properties")
@PropertySource("classpath:/vehicle-factory.properties")
class VehicleFactoryConfig {}
```

5.@PropertySources

We can use this annotation to specify multiple *@PropertySource* configurations:

```
@Configuration
@PropertySources({
    @PropertySource("classpath:/annotations.properties"),
    @PropertySource("classpath:/vehicle-factory.properties")
})
class VehicleFactoryConfig {}
```

Spring Validation

JSR-303 standardizes validation constraint declaration and metadata for the Java platform. Using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them

JSR-303 allows you to define declarative validation constraints against such properties:

```
public class Car {

   @NotNull
   private String manufacturer;

   @NotNull
   @Size(min = 2, max = 14)
   private String licensePlate;

   @Min(2)
   private int seatCount;

   // ...
}
```

When an instance of this class is validated by a JSR-303 Validator, these constraints will be enforced.

For general information on JSR-303/JSR-349, see the Bean Validation website. For information on the specific capabilities of the default reference implementation, see the Hibernate Validator documentation.

Spring Expression Language (SpEL)

The Spring Expression Language (SpEL) is a powerful expression language that supports querying and manipulating an object graph at runtime. It can be used with XML or annotation-based Spring configurations.

There are several operators available in the language:

Type	**Operators**
Arithmetic	+, -, *, /, %, ^, div, mod
Relational	<, >, ==, !=, <=, >=, lt, gt, eq, ne, le, ge
Logical	and, or, not, &&, ||, !
Conditional	?:
Regex	matches

Syntax

```
#{ Some Operation }
```

1.Arithmetic Operators

```
@Value("#{19 + 1}") // 20
private double add;

@Value("#{'String1 ' + 'string2'}") // "String1 string2"
private String addString;

@Value("#{20 - 1}") // 19
private double subtract;

@Value("#{10 * 2}") // 20
private double multiply;

@Value("#{36 / 2}") // 19
private double divide;

@Value("#{36 div 2}") // 18, the same as for / operator
private double divideAlphabetic;

@Value("#{37 % 10}") // 7
private double modulo;

@Value("#{37 mod 10}") // 7, the same as for % operator
private double moduloAlphabetic;

@Value("#{2 ^ 9}") // 512
private double powerOf;
```

```
@Value("#{(2 + 2) * 2 + 9}") // 17
private double brackets;
```

Relational and Logical Operators

```
@Value("#{1 == 1}") // true
private boolean equal;

@Value("#{1 eq 1}") // true
private boolean equalAlphabetic;

@Value("#{1 != 1}") // false
private boolean notEqual;

@Value("#{1 ne 1}") // false
private boolean notEqualAlphabetic;

@Value("#{1 < 1}") // false
private boolean lessThan;

@Value("#{1 lt 1}") // false
private boolean lessThanAlphabetic;

@Value("#{1 <= 1}") // true
private boolean lessThanOrEqual;

@Value("#{1 le 1}") // true
private boolean lessThanOrEqualAlphabetic;

@Value("#{1 > 1}") // false
private boolean greaterThan;

@Value("#{1 gt 1}") // false
private boolean greaterThanAlphabetic;

@Value("#{1 >= 1}") // true
private boolean greaterThanOrEqual;

@Value("#{1 ge 1}") // true
private boolean greaterThanOrEqualAlphabetic;
```

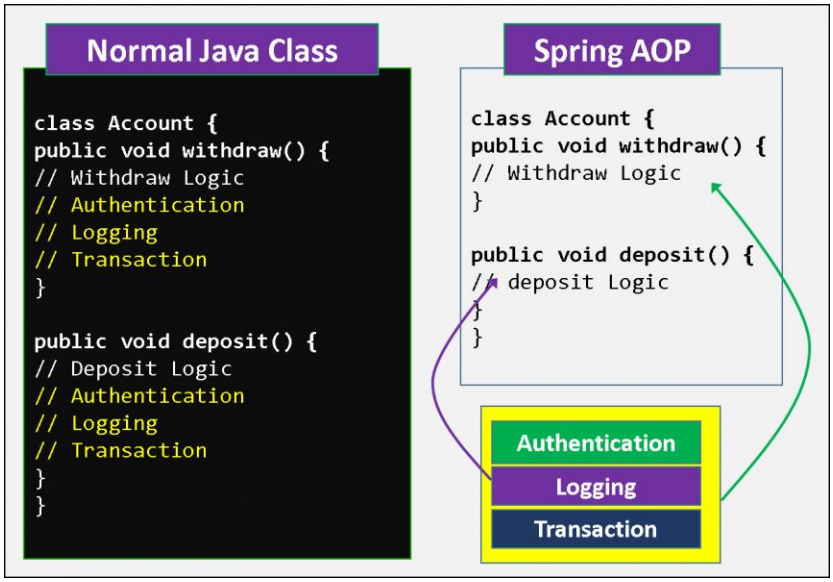
Using Regex in SpEL

```
@Value("#{'100' matches '\\d+' }") // true
private boolean validNumericStringResult;

@Value("#{'100fghdjf' matches '\\d+' }") // false
private boolean invalidNumericStringResult;

@Value("#{'valid alphabetic string' matches '[a-zA-Z\\s]+' }") // true
private boolean validAlphabeticStringResult;

@Value("#{'invalid alphabetic string #$1' matches '[a-zA-Z\\s]+' }") // false
private boolean invalidAlphabeticStringResult;

@Value("#{someBean.someValue matches '\d+'}") // true if someValue contains only digits
private boolean validNumericValue;
```

II. Spring AOP

Spring AOP (**Aspect-oriented programming**) framework is used for adding different cross-cutting functionalities. cross-cutting functionalities means adding different types of services to the application at runtime automatically.

In below Account class, we have **withdraw() & deposit()** methods each have Authentication, Logging, Trasaction cross-cutting functionalities. These are repeating in same class & also if we have 50 methods, we have to write these functionalities in 50 methods so, code is repeating.



In order to overcome the above problems, we need to separate the business logic and the services, is known as AOP, Using AOP the business logic and cross-cutting functionalities are implemented separately and executed at run time as combine.

AOP is a Specification, Spring framework is implemented it. AOP implementations are provided by

- Spring AOP, AspectJ, JBoss AOP

AOP Terminology

We use these 9 terminilogies very common on Spring AOP

1. **Aspect**
2. **Advice**
3. **JoinPoint**
4. **Pointcut**
5. **Introduction**
6. **Target**
7. **Proxy**

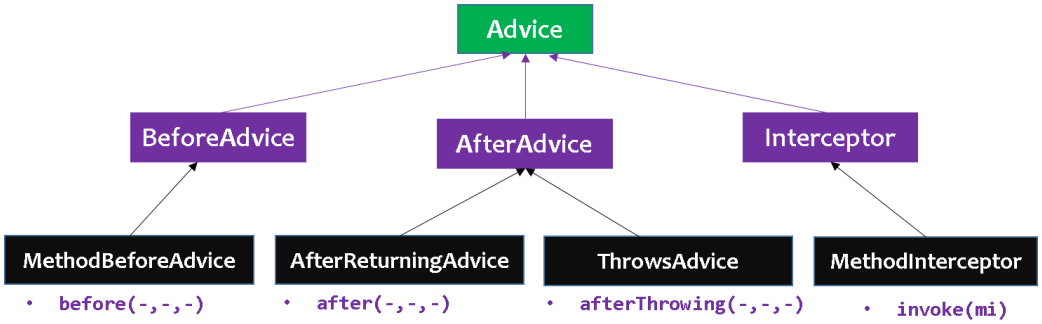
8. **Weaving**
9. **Adviser**

1. Aspect

- An aspect represents the cross-cutting functionality name, **just name only.**
- Aspect denotes **only the cross-cutting functionality name not its implementation**
- In above, we have 3 Aspects: **Authentication Aspect, Logging Aspect, Transaction Aspect.**

2. Advice

- Advice is **the implementation of Aspect**.
- An Advice provides the code for implementation of the service.
- The Implementation of above aspects called as **Authentication Advice, Logging Advice, Transaction Advice.**



In Spring AOP we have 4 **Types of Advices**

1.Before Advice

- services will be applied **before business logic**
- **MethodBeforeAdvice** interface extends the **BeforeAdvice** interface.
- If we implement MethodBeforeAdvice interface, we need to override **before()** method.
- before() method are executed at before business logic

```
public class  beforeAdvice implements MethodBeforeAdvice
{
    public void before(Method m,Object args[], Object target)throws Exception
    {
                //My Before Logic...
    }
}
```

2.After Advice

- services will be applied **After business logic**
- **AfterReturningAdvice** interface extends the **AfterAdvice** interface.
- we need to override **afterReturning()** method

```
public class  afterAdvice implements AfterReturningAdvice
{
public void afterReturning(Object retnVal,Object args[], Object target)throws Exception
    {
```

```
                //My Before Logic...
    }
}
```

3.Around Advice

- It is the **combination of both Before and After Advice.**
- **MethodInterceptor** interface extends the **Interceptor** interface**.**
- In Around Advice, we implement Before and After Advice in a single method called **invoke(),** in order to separate Before an After services to execute business logic, in the middle we call **proceed()** method

```
public class Client implements MethodInterceptor
{
    public Object invoke(MethodInvocation mi)throws Throwable
    {
        //Before Logic
        Object ob = mi.proceed();
        //After logic
        return ob;

    }
}
```

4.Throws Advice

- services will be applied **when business logic methods throws an exception.**
- **ThrowsAdvice** interface also extends the **AfterAdvice** interface.
- we should implement **afterThrowing()** method

```
public class Client implements ThrowsAdvice
{
   public void afterThrowing(Method m,Object args[],Object target,Exception e)
   {
       // our services
   }
}
```

3. JoinPoint

While creating the business logic of the method the **additional services are needed to be injected** at different places or points, we call such points as **joinpoints**. At a joinpoint a new service will be added into the normal flow of a business method.

While executing the business method, the services are required at the following **3** places, we call them as JoinPoints.

- Before business logic of the method starts
- After business logic of the method got completed
- If business logic throws an exception at run time

4. Pointcut

A pointcut defines what advices are required at what join points. In above diagram **Authentication Advice, Logging Advice, Transaction Advice** are required after withdraw logic & after balance logic. So this point is known as PointCut.

5. Introduction

It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

6. Target Object

It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

7. Aspect

It is a class that contains advices, joinpoints etc.

8. Interceptor

It is an aspect that contains only one advice.

9. AOP Proxy

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

10. Weaving

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

Spring AOP can be used by 3 ways given below. But the widely used approach is Spring AspectJ Annotation Style. The 3 ways to use spring AOP are given below:

- **By Spring1.2 Old style (dtd based)**
- **By AspectJ annotation-style & XML configuration-style**

Spring AOP –DTD based Example

1.Create Account.java class that contains actual business logic.

```
public class Account {
	private double balance;

	public double getBalance() {
		return balance;
	}
	public void setBalance(double balance) {
		this.balance = balance;
	}
	public void withdraw(double amt) {
		balance = balance - amt;
		System.out.println("Withdraw Complted.Bal is : " + balance);
	}
	public void deposite(double amt) {
		balance = balance + amt;
		System.out.println("Deposite Complted.Bal is : " + balance);
	}
}
```

2. create advisor classes that implements above 4 mentioned Advice interfaces

```
//file: BeforeAdviceEx.java
public class BeforeAdviceEx implements MethodBeforeAdvice {
	@Override
	public void before(Method m, Object[] args, Object target) throws Throwable {
	System.out.println("1.Before Adice : Executed ******");
	}
}
=============================================================================================
//file : AfterAdviceEx.java
public class AfterAdviceEx implements AfterReturningAdvice {
	@Override
	public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws
Throwable {
		System.out.println("2. AFTER Advice Executed *****");
	}
}
=============================================================================================
//file : AroundAdviceEx.java
public class AroundAdviceEx implements MethodInterceptor {
	@Override
	public Object invoke(MethodInvocation mi) throws Throwable {
		System.out.println("3.AROUND ADVICE ======");
		Object obj;
		System.out.println("----Before Business logic");
		obj = mi.proceed();
		System.out.println("----After Business logic");
		return obj;
	}
}
=============================================================================================
```

```
//file : ThrowsAdviceEx.java
public class ThrowsAdviceEx implements ThrowsAdvice {
        public void afterThrowing(java.lang.ArithmeticException ex){
        System.out.println("4.ThrowsAdvice : Error Occured!!!");
    }
}
```

3. Create SpringConfig.xml

- create beans for Account class, four Advisor classes and for **ProxyFactoryBean** class.
- **ProxyFactoryBean** class contains 2 properties **target** and **interceptorNames**.
    - **target**: The instance of Account class will be considered as target object.
    - **interceptorNames**: the instances of advisor classes. we need to pass the advisor object as the list object as in the xml file given above.

```
<!-- File : SpringConfig.xml -->
<beans>

        <bean id="acc" class="Account">
                <property name="balance" value="1000" />
        </bean>

        <bean id="beforeObj" class="BeforeAdviceEx"></bean>
        <bean id="afterObj" class="AfterAdviceEx"></bean>
        <bean id="aroundObj" class="AroundAdviceEx"></bean>
        <bean id="throwsObj" class="ThrowsAdviceEx"></bean>

        <bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="target" ref="acc"></property>
                <property name="interceptorNames">
                        <list>
                                <value>beforeObj</value>
                                <value>afterObj</value>
                                <value>aroundObj</value>
                                <value>throwsObj</value>
                        </list>
                </property>
        </bean>

</beans>
```

4.Create AOPTest.java for testing the Application

```
public class AOPTest {
public static void main(String[] args) {
         Resource res = new ClassPathResource("SpringConfig.xml");
     BeanFactory factory = new XmlBeanFactory(res);
     Account account = (Account) factory.getBean("proxy");
     account.deposite(500);
}
}
```

```
1.Before Adice : Executed ******
3.AROUND ADVICE ======
----Before Business logic
Deposite Complted.Bal is : 1500.0
----After Business logic
2. AFTER Advice Executed *****
```

Note : here we are getting "proxy" bean object to apply AOP to the application

he **Spring Framework** recommends you to use **Spring AspectJ AOP implementation** over the Spring 1.2 old style dtd based AOP implementation because it provides you more control and it is easy to use.

There are two ways to use Spring AOP AspectJ implementation:

- **By annotation**
- **By xml configuration**

1. AspectJ –By Annotations

Common AspectJ annotations

1. **@Aspect** declares the class as aspect.
2. **@Pointcut** declares the pointcut expression.
3. **@Before** – Run before the method execution
4. **@After** – Run after the method returned a result
5. **@AfterReturning** – Run after the method returned a result, intercept the returned result as well.
6. **@AfterThrowing** – Run after the method throws an exception
7. **@Around** – Run around the method execution, combine all three advices above.

Pointcut: Pointcut is an expression language of Spring AOP.The **@Pointcut** annotation is used to define the pointcut. We can refer the pointcut expression by name also. Let's see the simple example of pointcut expression.

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

1.Student.java: Normal bean, with few methods, StudentImpl its implementation class

```
//file :Student.java
public interface Student {

	void addStudent();
	String studentReturnValue();
	void studentThrowException() throws Exception;
	void studentAround(String name);

}
```

```
//file :StudentImpl.java
public class StudentImpl implements Student {
	@Override
	public void addStudent() {
		System.out.println("Satya : new Student Added");
	}
	@Override
	public String studentReturnValue() {
		return "Return Student: satya";
	}

	@Override
```

```
	public void studentThrowException() throws Exception {
		System.out.println("studentThrowException() is running ");
		throw new Exception("Student Error");
	}

	@Override
	public void studentAround(String name) {
		System.out.println("studentAround() is running, args : " + name);
	}
}
```

2. SpringConfig.xml: put "<aop:aspectj-autoproxy />", and define Aspect & normal bean.

```
<beans>
	<aop:aspectj-autoproxy />
	<bean id="studentOb" class="StudentImpl" />

	<!-- Aspect -->
	<bean id="logAspect" class="LoggingAspect" />
</beans>
```

3. Write Aspect class to Apply asepcts & define PointCut's where to apply those aspects

AspectJ "pointcuts" is used to declare which method is going to intercept.

```
// LoggingAspect.java
@Aspect
public class LoggingAspect {

	@Before("execution(* addStudent(..))")
	public void logBefore(JoinPoint joinPoint) {

		System.out.println("logBefore() is running!");
		System.out.println(joinPoint.getSignature().getName());
		System.out.println("******");
	}

	@After("execution(* addStudent(..))")
	public void logAfter(JoinPoint joinPoint) {

		System.out.println("logAfter() is running!");
		System.out.println(joinPoint.getSignature().getName());
		System.out.println("******");

	}

	@AfterReturning(
			pointcut = "execution(* studentReturnValue(..))",
			returning= "result")
	public void logAfterReturning(JoinPoint joinPoint, Object result) {

		System.out.println("logAfterReturning() is running!");
		System.out.println(joinPoint.getSignature().getName());
		System.out.println("Method returned value is : " + result);
		System.out.println("******");

	}

	@AfterThrowing(
			pointcut = "execution(* studentThrowException(..))",
			throwing= "error")
	public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {
```

```
                System.out.println("logAfterThrowing() is running!");
                System.out.println(joinPoint.getSignature().getName());
                System.out.println("Exception : " + error);
                System.out.println("******");

        }


        @Around("execution(* studentAround(..))")
        public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {

                System.out.println("logAround() is running!");
                System.out.println("method : " + joinPoint.getSignature().getName());
                System.out.println("arguments : " + Arrays.toString(joinPoint.getArgs()));

                System.out.println("Around before is running!");
                joinPoint.proceed();
                System.out.println("Around after is running!");

                System.out.println("******");

        }
}
```

4.Test Class to test the Application

```
public class AspectJTestApp {
        public static void main(String[] args) throws Exception {

                 Resource res = new ClassPathResource("SpringConfig.xml");
             BeanFactory factory = new XmlBeanFactory(res);

                Student s = (Student) factory.getBean("studentOb");
                s.addStudent();
                s.studentReturnValue();
                s.studentAround("SATYA");
                s.studentThrowException();

        }
}
```

```
Exception in thread "main" Satya : new Student Added
studentAround() is running, args : SATYA
studentThrowException() is running
java.lang.Exception: Student Error
        at StudentImpl.studentThrowException(StudentImpl.java:18)
        at AspectJTestApp.main(AspectJTestApp.java:20)
```

2. AspectJ –By XML Configuration

Let's see the xml elements that are used to define advice.

- **<aop:before>** **= @Before**
- **<aop:after>** **= @After**

- **<aop:after-returning> = @AfterReturning**
- **<aop:after-throwing> = @AfterThrowing**
- **<aop:after-around> = @Around**

In this example Student, StudentImpl,LoggingAspect java files are same as Annotation Example

1.LoggingAspect.java

```
@Aspect
public class LoggingAspect {
	public void logBefore(JoinPoint joinPoint) {
		System.out.println("logBefore() is running!");
		System.out.println(joinPoint.getSignature().getName());
		System.out.println("******");
	}

	public void logAfter(JoinPoint joinPoint) {
		System.out.println("logAfter() is running!");
		System.out.println(joinPoint.getSignature().getName());
		System.out.println("******");
	}

	public void logAfterReturning(JoinPoint joinPoint, Object result) {
		System.out.println("logAfterReturning() is running!");
		System.out.println(joinPoint.getSignature().getName());
		System.out.println("Method returned value is : " + result);
		System.out.println("******");
	}

	public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {
		System.out.println("logAfterThrowing() is running!");
		System.out.println(joinPoint.getSignature().getName());
		System.out.println("Exception : " + error);
		System.out.println("******");
	}

	public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {
		System.out.println("logAround() is running!");
		System.out.println("method : " + joinPoint.getSignature().getName());
		System.out.println("arguments : " + Arrays.toString(joinPoint.getArgs()));

		System.out.println("Around before is running!");
		joinPoint.proceed();
		System.out.println("Around after is running!");
		System.out.println("******");
	}
}
```

SpringConfig.java

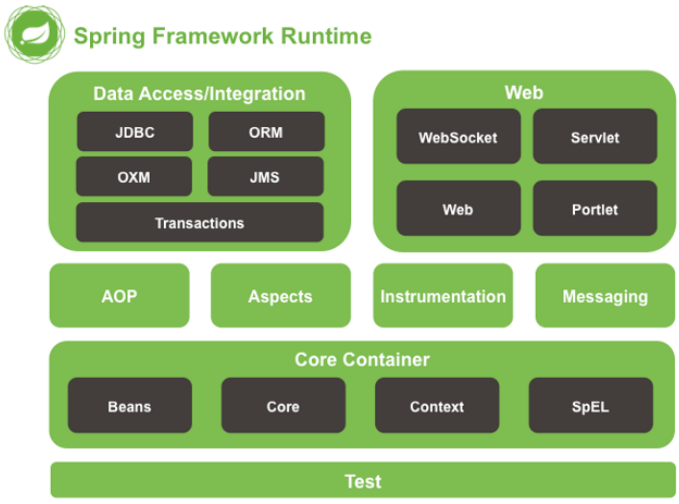
```
<beans>
	<aop:aspectj-autoproxy />

	<bean id="studentOb" class="StudentImpl" />

	<!-- Aspect -->
	<bean id="logAspect" class="LoggingAspect" />

<aop:config>
	<aop:aspect id="aspectLoggging" ref="logAspect">

		<!-- @Before -->
```

```
                <aop:pointcut id="pointCutBefore" expression="execution(* addStudent(..))" />
                <aop:before method="logBefore" pointcut-ref="pointCutBefore" />

                <!-- @After -->
                <aop:pointcut id="pointCutAfter" expression="execution(* addStudent(..))" />
                <aop:after method="logAfter" pointcut-ref="pointCutAfter" />

                <!-- @AfterReturning -->
                <aop:pointcut id="pointCutAfterReturning" expression="execution(* studentReturnValue(..))"
/>
                <aop:after-returning method="logAfterReturning"
                returning="result" pointcut-ref="pointCutAfterReturning" />


                <!-- @AfterThrowing -->
                <aop:pointcut id="pointCutAfterThrowing"
                                expression="execution(* studentThrowException(..))" />
                <aop:after-throwing method="logAfterThrowing"
                                throwing="error" pointcut-ref="pointCutAfterThrowing" />


                <!-- @Around -->
                <aop:pointcut id="pointCutAround" expression="execution(* studentAround(..))" />
                <aop:around method="logAround" pointcut-ref="pointCutAround" />

        </aop:aspect>
</aop:config>
</beans>
```

```
public class AspectJTestApp {
        public static void main(String[] args) throws Exception {
                 Resource res = new ClassPathResource("SpringConfig.xml");
             BeanFactory factory = new XmlBeanFactory(res);

                Student s = (Student) factory.getBean("studentOb");
                s.addStudent();
                s.studentReturnValue();
                s.studentAround("SATYA");
                s.studentThrowException();
        }
}
```

```
Satya : new Student Added
Exception in thread "main" studentAround() is running, args : SATYA
studentThrowException() is running
java.lang.Exception: Student Error
       at StudentImpl.studentThrowException(StudentImpl.java:17)
       at AspectJTestApp.main(AspectJTestApp.java:20)
```

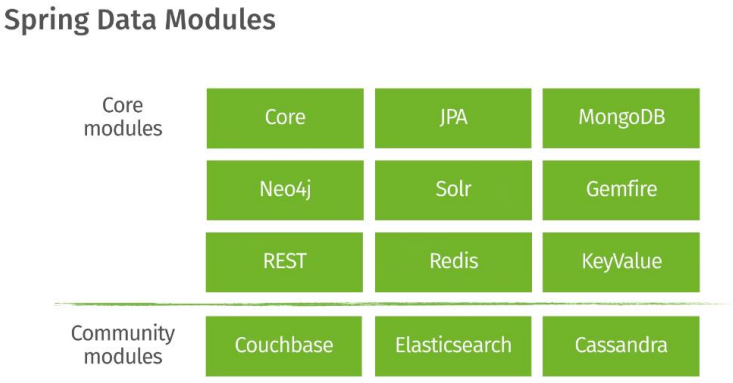
III.Spring Data Access



When Spring Framework was created, in early 2000s, the only kind of database was relational database - Oracle, MS SQL Server, My SQL etc. In the last few years, there are a wide variety of databases that are getting popular - most of them not relational and not using SQL. Wide variety of terminology is used to refer to these databases. NoSQL, for example.

ORM frameworks (Hibernate) and specifications(JPA) were good fit for the relational databases. But, the newer databases, have different needs

Main modules



- Spring Data JDBC - Spring Data repository support for JDBC.
- Spring Data JPA - Spring Data repository support for JPA.
- Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB
- Spring Data LDAP - Spring Data repository support for Spring LDAP.
- Spring Data REST - Exports Spring Data repositories as hypermedia-driven RESTful resources.

Spring JDBC

The Spring Data access logic revolves around ***Template*** patterns and **Support** classes

Drawbacks of JDBC

- In JDBC all the **exceptions are checked**, so we must use try, catch blocks in the code.
- if we **open the connection** with database, **we only responsible to close that connection.**
- JDBC error messages are Database related error messages, not every one may understand.

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

We have following templates to work with JDBC related things in Spring.

- **JdbcTemplate**
- **NamedParameterJdbcTemplate**
- **SimpleJdbcTemplate**

JdbcTemplate class

- **JdbcTemplate** class is given **in org.springframework.jdbc.core.*** package and this class will provide methods for executing the SQL commands on a database

- JdbcTemplate class **follows template design pattern**, where a template class accepts input from the user and produces output to the user by hiding the interval details

Method	Description
public void execute(String query)	is used to execute DDL query.
public int update(String query)	is used to insert, update and delete records.
List queryForInt("query") **List queryForObject("query")** **List queryForXXX("query")**	For selecting the records from Database.
public T execute(String sql, PreparedStatementCallback action)	executes the query by using PreparedStatement callback.
public T query(String sql, ResultSetExtractor rse)	is used to fetch records using ResultSetExtractor.
public List query(String sql, RowMapper rse)	is used to fetch records using RowMapper.

JdbcTemplate: Simple SQL Statements Example

1.select Database

```
CREATE TABLE `student` (
`sno` INT(11) NOT NULL AUTO_INCREMENT,
`name` VARCHAR(50) NULL DEFAULT NULL,
`address` VARCHAR(50) NULL DEFAULT NULL,
PRIMARY KEY (`sno`)
);
```

2.Student.java

This class contains 3 properties with constructors and setter and getters.

```
//File: Student.java
public class Student {
	private int sno;
	private String name;
	private String address;

	public Student() {
		super();
	}
	public Student(int sno, String name, String address) {
		this.sno = sno;
		this.name = name;
		this.address = address;
	}
	//Setters & getters
}
```

3.StudentDao

- **JdbcTemplate** class executes SQL queries or updates, initiating iteration over ResultSet and catching JDBC exceptions and translating.
- To call JdbcTemplate methods, we need initialize JdbcTemplate object in our DAO class.
- For that we declared JdbcTemplate property in our StudentDao class & will inject JdbcTemplate object from SpringConfig.xml file

```
package jdbc;

import java.util.Iterator;
import java.util.List;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentDao {
	private JdbcTemplate jdbcTemplate;

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

	public int saveStudent(Student s) {
		String query = "insert into student values('" + s.getSno() +
"','" + s.getName() + "','" + s.getAddress()
				+ "')";
		return jdbcTemplate.update(query);
	}
```

```
	public int updateStudent(Student s) {
		String query = "update student set name='" + s.getName() +
"',address='" + s.getAddress() + "' where sno='"
				+ s.getSno() + "' ";
		return jdbcTemplate.update(query);
	}

	public int deleteStudent(Student s) {
		String query = "delete from student where sno='" + s.getSno() +
"' ";
		return jdbcTemplate.update(query);
	}

	public void selectStudents() {
		List l = jdbcTemplate.queryForList("select * from student");
		Iterator it = l.iterator();
		while (it.hasNext()) {
			Object o = it.next();
			System.out.println(o.toString());
		}
	}
}
```

4. SpringConfig.java

We have to configure 3 properties in SpringConfig.xml. they are

1. Create DataSource object

- Spring-JDBC, **the programmer no need to open and close the database connection** and it will be taken care by the spring framework.

- Spring framework **uses DataSource interface** to obtain the connection with database internally.

- will use any one of the following **2** implementation classes of **DataSource** interface

```
org.springframework.jdbc.datasource.DriverManagerDataSource
org.apache.commons.dbcp.BasicDataSource
```

- We have to provide connection details to DataSource object

```
<!-- 1. Creating DataSource object  -->
<bean id="ds"    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/smlcodes" />
        <property name="username" value="root" />
        <property name="password" value="root" />
</bean>
```

2. Create JdbcTemplate

JdbcTemplate class depends on **DataSource** object only, as it will open database connection internally with DataSource. So we must give this DataSource object to JdbcTemplate.

```
        <!-- 2. Creating JdbcTemplate object  -->
```

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

3. Inject JdbcTemplate object to StudentDao class property.

File : SpringConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
        <!-- 1. Creating DataSource object  -->
        <bean id="ds"    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/smlcodes" />
                <property name="username" value="root" />
                <property name="password" value="root" />
        </bean>

        <!-- 2. Creating JdbcTemplate object  -->
        <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource" ref="ds"></property>
        </bean>
        <!-- 3. Injecting JdbcTemplate object to StudentDao class property  -->
        <bean id="dao" class="jdbc.StudentDao">
                <property name="jdbcTemplate" ref="jdbcTemplate"></property>
        </bean>
</beans>
```

JdbcTestApplication.java

```
//File: JdbcTestApplication.java
public class JdbcTestApplication {
      public static void main(String[] args) {
            Resource res = new ClassPathResource("jdbc/SpringConfig.xml");
            BeanFactory factory = new XmlBeanFactory(res);

            System.out.println("1.INSERT \n ------------");
            StudentDao dao = (StudentDao) factory.getBean("dao");
            Student s = new Student(102, "Satya", "HYDERABAD");
            int r = dao.saveStudent(s);
            System.out.println(r + " Records are Effected");

            System.out.println(" \n 2.SELECT \n ------------");
            dao.selectStudents();

            System.out.println(" \n 3.UPDATE \n ------------");
            s.setName("RAVI");
            dao.updateStudent(s);
            dao.selectStudents();

            System.out.println(" \n 4.DELETE \n ------------");
            dao.deleteStudent(s);
            dao.selectStudents();
      }
}
```

```
1.INSERT
 ------------
1 Records are Effected

 2.SELECT
 ------------
{sno=102, name=Satya, address=HYDERABAD}
```

```
 3.UPDATE
 ------------
{sno=102, name=RAVI, address=HYDERABAD}

 4.DELETE
 ------------
```

PreparedStatementCallback

We can execute parameterized query using Spring JdbcTemplate by the help of **execute()** method of JdbcTemplate class. To use parameterized query, we pass the instance of **PreparedStatementCallback** in the execute method.

```
public T execute (String sql,PreparedStatementCallback<T>);
```

It has only one method **doInPreparedStatement(PreparedStatement ps)**

JdbcTemplate-PreparedStatement Example

```
//File : StudentPreparedStmntDao.java
public class StudentPreparedStmntDao {
	private JdbcTemplate jdbcTemplate;

	public JdbcTemplate getJdbcTemplate() {
		return jdbcTemplate;
	}

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

	public Boolean saveStudentByPreparedStatement(final Student s) {
		String query = "insert into student values(?,?,?)";
		return jdbcTemplate.execute(query, new PreparedStatementCallback<Boolean>() {
			@Override
			public Boolean doInPreparedStatement(PreparedStatement ps) throws SQLException,
DataAccessException {

				ps.setInt(1, s.getSno());
				ps.setString(2, s.getName());
				ps.setString(3, s.getAddress());
				return ps.execute();
			}
		});
	}
}
```

```
//File : PreparedStmtTestApplication.java
public class PreparedStmtTestApplication {
	public static void main(String[] args) {
		Resource res = new ClassPathResource("jdbc/SpringConfig.xml");
		BeanFactory factory = new XmlBeanFactory(res);

		StudentPreparedStmntDao dao = (StudentPreparedStmntDao) factory.getBean("dao");
		Student s = new Student(102, "Satya", "HYDERABAD");
		boolean r = dao.saveStudentByPreparedStatement(s);
		System.out.println(" Data Inserted : "+r);
	}
}
```

```
<!-- File : SpringConfig.xml -->
<beans>
	<bean id="ds"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName" value="com.mysql.jdbc.Driver" />
		<property name="url" value="jdbc:mysql://localhost:3306/smlcodes" />
		<property name="username" value="root" />
		<property name="password" value="root" />
	</bean>

	<!-- 1. Creating JdbcTemplate object  -->
	<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
		<property name="dataSource" ref="ds"></property>
	</bean>
	<!-- 2. Injecting JdbcTemplate object to StudentDao class property  -->
	<bean id="dao" class="jdbc.StudentPreparedStmntDao">
		<property name="jdbcTemplate" ref="jdbcTemplate"></property>
	</bean>
</beans>
```

3. ResultSetExtractor

We can easily fetch the records from the database using **query()** method of **JdbcTemplate** class where we need to pass the instance of ResultSetExtractor.

```
public T query(String sql,ResultSetExtractor<T> rse)
```

It defines only one method **public** T **extractData(ResultSet rs)** that accepts ResultSet instance as a parameter

ResultSetExtractor Fetching Records Example

```
//File : StudentPreparedStmntDao.java
public class StudentPreparedStmntDao {
	private JdbcTemplate jdbcTemplate;

	public JdbcTemplate getJdbcTemplate() {
		return jdbcTemplate;
	}

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}
	public List<Student> getAllstudents() {
		return jdbcTemplate.query("select * from student", new
ResultSetExtractor<List<Student>>() {
			@Override
			public List<Student> extractData(ResultSet rs) throws
SQLException, DataAccessException {

				List<Student> list = new ArrayList<Student>();
				while (rs.next()) {
					Student e = new Student();
					e.setSno(rs.getInt(1));
					e.setName(rs.getString(2));
					e.setAddress(rs.getString(3));
					list.add(e);
				}
```

```
                        return list;
                    }
                });
        }
}
```

```
//File : PreparedStmtTestApplication.java
public class PreparedStmtTestApplication {
        public static void main(String[] args) {
                Resource res = new ClassPathResource("jdbc/SpringConfig.xml");
                BeanFactory factory = new XmlBeanFactory(res);

                StudentPreparedStmntDao dao = (StudentPreparedStmntDao) factory.getBean("dao");
                List<Student> list = dao.getAllstudents();
                for (Student e : list)
                        System.out.println(e);
                }
}
```

NamedParameterJdbcTemplate class

Spring provides another way to insert data by named parameter. In such way, **we use names instead of? (question mark),** like below

```
insert into student values (:sno,:name,:address)
```

NamedParameterJdbcTemplate Example

```
public class StudentDao {
        private NamedParameterJdbcTemplate jdbcTemplate;

        public StudentDao(NamedParameterJdbcTemplate jdbcTemplate) {
                super();
                this.jdbcTemplate = jdbcTemplate;
        }

        public void saveStudent(Student e) {
                String query = "insert into Student values (:sno,:name,:address)";

                Map<String, Object> map = new HashMap<String, Object>();
                map.put("sno", e.getSno());
                map.put("name", e.getName());
                map.put("address", e.getAddress());

                jdbcTemplate.execute(query, map, new PreparedStatementCallback() {
                        @Override
                        public Object doInPreparedStatement(PreparedStatement ps) throws SQLException,
DataAccessException {
                                return ps.executeUpdate();
                        }
                });
        }
}
```

```
<!-- File : SpringConfig.xml -->
<beans>
        <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/smlcodes" />
                <property name="username" value="root" />
                <property name="password" value="root" />
        </bean>
```

```
        <bean id="jtemplate"
                class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
                <constructor-arg ref="ds"></constructor-arg>
        </bean>

        <bean id="dao" class="StudentDao">
                <constructor-arg>
                        <ref bean="jtemplate" />
                </constructor-arg>
        </bean>
</beans>
```

```
//File: JdbcTestApplication.java
public class JdbcTestApplication {
        public static void main(String[] args) {
                Resource res = new ClassPathResource("SpringConfig.xml");
                BeanFactory factory = new XmlBeanFactory(res);

                StudentDao dao = (StudentDao) factory.getBean("dao");
                Student s = new Student(103, "KAVETI", "HYDERABAD");
                dao.saveStudent(s);
        }
}
```

SimpleJdbcTemplate

Spring 3 JDBC supports the java 5 feature var-args (variable argument) and autoboxing by the help of SimpleJdbcTemplate class. SimpleJdbcTemplate class wraps the JdbcTemplate class and provides the update method where we can pass arbitrary number of arguments

```
int update(String sql,Object... parameters)
```

here We should pass the parameter values in the update method in the order they are defined in the parameterized query

```
//File : StudentDao.java
public class StudentDao {
        private SimpleJdbcTemplate jdbcTemplate;
        public StudentDao(SimpleJdbcTemplate jdbcTemplate) {
                this.jdbcTemplate = jdbcTemplate;
        }

        public int updateStudent(Student e) {
                String query = "update student set name=? where sno=?";
                return jdbcTemplate.update(query, e.getName(), e.getSno());
        }
}
```

```
//SpringConfig.xml
<bean id="jtemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
        <constructor-arg ref="ds"></constructor-arg>
</bean>
```

```
//File: JdbcTestApplication.java
public class JdbcTestApplication {
        public static void main(String[] args) {
                Resource res = new ClassPathResource("SpringConfig.xml");
                BeanFactory factory = new XmlBeanFactory(res);
```

```
                StudentDao dao = (StudentDao) factory.getBean("dao");
                Student s = new Student(103, "RAM", "HYDERABAD");
                int i =dao.updateStudent(s);
                System.out.println(i);
        }
}
```

We have JPA & Hibernate integations in this topic. We will cover these things in Spring Integration with Other frameworks topic ☺

Spring ORM

JPA Example

Mapping Java objects to database tables and vice versa is called *Object-relational mapping*(ORM).
The Java Persistence API (JPA) is one possible approach to ORM.

- **JPA is a specification** and several implementations are available. Popular implementations **are Hibernate, EclipseLink and Apache OpenJPA.**
- JPA permits the developer to work directly with objects rather than with SQL statements.
- The mapping between Java objects and database tables is defined via persistence metadata. **JPA metadata is typically defined via annotations or xml files.**

Spring Data JPA API provides **JpaTemplate** class to integrate spring application with JPA.

1.Student.java : It is a simple POJO class

```
package smlcodes;

public class Student {
        private int sno;
        private String name;
        private String address;
        public int getSno() {
                return sno;
        }
        public void setSno(int sno) {
                this.sno = sno;
        }
        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 Student(int sno, String name, String address) {
                super();
                this.sno = sno;
                this.name = name;
```

```
                this.address = address;
        }

        public Student() {
                super();
        }
}
```

2.Student.xml: This mapping file contains all the information of the persistent class

```
<entity-mappings version="1.0"
        xmlns="http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                http://java.sun.com/xml/ns/persistence/orm_1_0.xsd ">

        <entity class="smlcodes.Student">
                <table name="student"></table>
                <attributes>
                        <id name="sno">
                                <column name="sno" />
                        </id>
                        <basic name="name">
                                <column name="name" />
                        </basic>
                        <basic name="address">
                                <column name="address" />
                        </basic>
                </attributes>
        </entity>
</entity-mappings>
```

3.StudentDao.java : DAO Class

```
@Transactional
public class StudentDao {
        JpaTemplate template;

        public void setTemplate(JpaTemplate template) {
                this.template = template;
        }

        public void saveStudent(int sno, String name, String address) {
                Student student = new Student(sno, name, address);
                template.persist(student);
        }

        public void updateStudent(int sno, String name) {
                Student student = template.find(Student.class, sno);
                if (student != null) {
                        student.setName(name);
                }
                template.merge(student);
        }

        public void deleteStudent(int sno) {
                Student student = template.find(Student.class, sno);
                if (student != null)
                        template.remove(student);
        }

        public List<Student> getAllStudents() {
                List<Student> students = template.find("select s from student s");
                return students;
        }
}
```

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
        <persistence-unit name="jppPU" transaction-type="RESOURCE_LOCAL">
                <mapping-file>smlcodes/Student.xml</mapping-file>
                <class>smlcodes.Student</class>
        </persistence-unit>
</persistence>
```

SpringConfig.xml

```
<beans>
 <tx:annotation-driven transaction-manager="jpaTxnManagerBean" proxy-target-class="true"/>
<bean id="dataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/springdb" />
                <property name="username" value="root" />
                <property name="password" value="root" />
</bean>


<bean id="emfBean" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
                <property name="dataSource" ref="dataSourceBean"></property>
                <property name="jpaVendorAdapter" ref="hbAdapterBean"></property>
         </bean>

         <bean id="jpaTemplateBean" class="org.springframework.orm.jpa.JpaTemplate">
                <property name="entityManagerFactory" ref="emfBean"></property>
         </bean>

         <bean id="studentsDaoBean" class="smlcodes.StudentDao">
                <property name="template" ref="jpaTemplateBean"></property>
         </bean>
        <bean id="jpaTxnManagerBean" class="org.springframework.orm.jpa.JpaTransactionManager">
                <property name="entityManagerFactory" ref="emfBean"></property>
        </bean>
</beans>
```

StudentJPAExample.java

```
public class StudentJPAExample {
        public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("SpringConfig.xml");
                StudentDao studentsDao = context.getBean("studentsDaoBean", StudentDao.class);

                studentsDao.saveStudent(101, "Satyar", "HYDERABAD");
                studentsDao.saveStudent(105, "RAJESH", "BANGLORE");
                System.out.println("Students created");

                studentsDao.updateStudent(105, "KARTHICK");
                System.out.println("Student Name updated");

                List<Student> students = studentsDao.getAllStudents();
                for (Student s : students) {
                System.out.println(s.getSno() + " : " + s.getName() + " , " + s.getAddress());
                }
                studentsDao.deleteStudent(111);
                System.out.println("Student deleted");
        }
}
```

Hibernate Example

The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

Commonly used methods of HibernateTemplate class.

Method	Description
void **persist(Object entity)**	persists the given object.
Serializable **save(Object entity)**	persists the given object and returns id.
void **saveOrUpdate(Object entity)**	persists or updates the given object. If id is found, it updates the record otherwise saves the record.
void **update(Object entity)**	updates the given object.
void **delete(Object entity)**	deletes the given object on the basis of id.
Object **get(Class entityClass, Serializable id)**	returns the persistent object on the basis of given id.
Object **load(Class entityClass, Serializable id)**	returns the persistent object on the basis of given id.
List **loadAll(Class entityClass)**	returns the all the persistent objects.

Student.java: It is a simple POJO class. Here it works as the persistent class for hibernate.

```
package smlcodes;

public class Student {
        private int sno;
        private String name;
        private String address;
//Setters & getters
}
```

Student.hbm.xml: This mapping file contains all the information of the persistent class.

```
<hibernate-mapping>
        <class name="smlcodes.Student" table="student">
                <id name="sno">
                        <generator class="assigned"></generator>
                </id>

                <property name="name"></property>
                <property name="address"></property>
        </class>
</hibernate-mapping>
```

StudentDao.java:it uses the **HibernateTemplate** class method to persist the object of Student class.

```
package smlcodes;

import org.springframework.orm.hibernate3.HibernateTemplate;

public class StudentDao {
        HibernateTemplate template;

        public void setTemplate(HibernateTemplate template) {
```

```
                this.template = template;
        }

        public void saveEmployee(Student e) {
                template.save(e);
        }

        public void updateEmployee(Student e) {
                template.update(e);
        }

        public void deleteEmployee(Student e) {
                template.delete(e);
        }
}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/springdb" />
                <property name="username" value="root" />
                <property name="password" value="root" />
        </bean>

        <bean id="mysessionFactory"
                class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
                <property name="dataSource" ref="dataSource"></property>
                <property name="mappingResources">
                        <list>  <value>student.hbm.xml</value></list>
                </property>

                <property name="hibernateProperties">
                        <props>
                        <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                                <prop key="hibernate.hbm2ddl.auto">update</prop>
                                <prop key="hibernate.show_sql">true</prop>
                        </props>
                </property>
        </bean>

        <bean id="template" class="org.springframework.orm.hibernate3.HibernateTemplate">
                <property name="sessionFactory" ref="mysessionFactory"></property>
        </bean>

        <bean id="d" class="smlcodes.StudentDao">
                <property name="template" ref="template"></property>
        </bean>
</beans>
```

In this file, we are providing all the information's of the database in the **BasicDataSource** object. This object is used in the **LocalSessionFactoryBean** class object, containing some other information's such as mappingResources and hibernateProperties. The object of **LocalSessionFactoryBean** class is used in the HibernateTemplate class. Let's see the code of applicationContext.xml file.

StudentHibernateExample.java

```
package smlcodes;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class StudentHibernateExample {
```

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

                Resource r = new ClassPathResource("applicationContext.xml");
                BeanFactory factory = new XmlBeanFactory(r);

                StudentDao dao = (StudentDao) factory.getBean("d");

                Student e = new Student();
                e.setSno(147);
                e.setName("kumar");
                e.setAddress("Hyderabad");

                dao.saveEmployee(e);
                // dao.updateEmployee(e);
        }
}
```

Remebember: We need to add Hibernate jars as well in this application.

Spring Transaction

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all.

The concept of transactions can be described with the following four key properties described as **ACID** −

- **Atomicity** − A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency** − This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
- **Isolation** − There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
- **Durability** − Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

Type of Transaction Management

In J2EE, Transaction Management can be divided in two types.

Global Transaction

Local Transaction

Global Transaction

- Use to work with multiple transaction resources like RDBMS or Message Queue (Pros)
- Managed by Application Server (WebSphere, Weblogic) using JTA (Cons)
- JNDI is required to use JTA
- Code can not be reused as JTA is available at server level(Cons)

- Example of Global Transaction: EJB CMT

Local Transaction

- Use to work with specific resource (transaction associated with JDBC)
- Can not work across multiple transaction resource opposite to Global transaction (cons)
- Most of web application uses only single resources hence it is best option to use in normal app

Approach for transaction management

Spring supports two different approach for transaction management.

Programmatic Transaction Management

Here you will **write code for transaction management**. Spring API dependency. Not good for maintenance. Good for development. Flexibity.

Declarative Transaction Management

Here you will **use XML or annotation for transaction management**. Less flexible but preferable over programmatic approach. In normal case no code is required for transaction management.

Programmatic Approch : using Java Classes

The key to the Spring transaction abstraction is defined by the *org.springframework.transaction.PlatformTransactionManager* interface, which is as follows –

PlatformTransactionManager

```
public interface PlatformTransactionManager {
   TransactionStatus getTransaction(TransactionDefinition definition);
   throws TransactionException;

   void commit(TransactionStatus status) throws TransactionException;
   void rollback(TransactionStatus status) throws TransactionException;
}
```

TransactionDefinition

```
public interface TransactionDefinition {
   int getPropagationBehavior();
   int getIsolationLevel();
   String getName();
   int getTimeout();
   boolean isReadOnly();
}
```

TransactionStatus

The *TransactionStatus* interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
   boolean isNewTransaction();
   boolean hasSavepoint();
   void setRollbackOnly();
   boolean isRollbackOnly();
   boolean isCompleted();
}
```

We have to use above classes to do Programatic Transaction management

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
// explicitly setting the transaction name is something that can only be done
programmatically
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
  // execute your business logic here
}
catch (MyException ex) {
  txManager.rollback(status);
  throw ex;
}
txManager.commit(status);
```

Declarative Approch : Using Annotations & XML

To start using **@Transactional** annotation in a Spring based application, we need to first enable annotations in our Spring application by adding the needed configuration into spring context file –

```
1 <tx:annotation-driven transaction-manager="txManager"/>
```

Next is to define the transaction manager bean, with the same name as specified in the above **transaction-manager**attribute value.

We have following types of transaction managers based upon the framework we use

For Simple JDBC

```
<bean id="txManager"
                class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref= "datasource" />
</bean>
```

For Hibernate

```
<bean id="txManager"
                class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref= "sessionFactory" />
</bean>
```

For JPA

```
<bean id="txManager"
                class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

We are now ready to use **@Transactional** annotation either at the class or method level

```
@Transactional(value = "myTransactionManager", propagation = Propagation.REQUIRED, readOnly = true)
public void myMethod() {
        ...
}
```

```
public class Customer {

        private int id;
        private String name;
        private Address address;
}
```

```
public class Address {

        private int id;
        private String city;
        private String country;
}
```

```
package prog;

import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.transaction.annotation.Transactional;

public class CustomerDAO {

        private DataSource dataSource;

        public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        @Transactional
        public void create(Customer customer) {
                String queryCustomer = "insert into Customer (id, name) values (?,?)";
                String queryAddress = "insert into Address (id, city,country) values (?,?,?)";

                JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
```

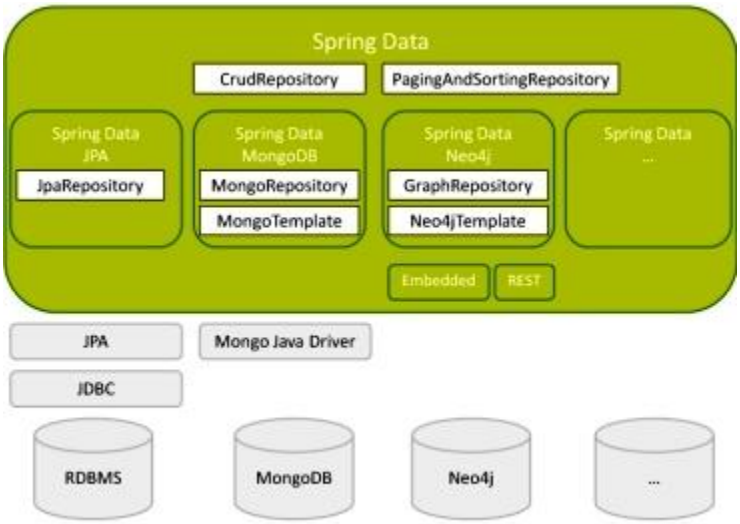
```
                jdbcTemplate.update(queryCustomer, new Object[] { customer.getId(), customer.getName() });
                System.out.println("Inserted into Customer Table Successfully");

                jdbcTemplate.update(queryAddress,
                                new Object[] { customer.getId(), customer.getAddress().getCity(),
customer.getAddress().getCountry() });
                System.out.println("Inserted into Address Table Successfully");
        }

}
```

```
package prog;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

        public static void main(String[] args) {
                ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

                CustomerDAO dao = context.getBean("customerDAO", CustomerDAO.class);

                Address address = new Address();
                address.setId(2);
                address.setCountry("India");
                address.setCity("HYD");

                Customer customer = new Customer();
                customer.setId(2);
                customer.setName("Pankaj");
                customer.setAddress(address);

                dao.create(customer);
                context.close();
        }
}
```

```
INFO: Loaded JDBC driver: com.mysql.jdbc.Driver
Inserted into Customer Table Successfully
Inserted into Address Table Successfully
```

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

        <!-- Enable Annotation based Declarative Transaction Management -->
        <tx:annotation-driven proxy-target-class="true"
                transaction-manager="transactionManager" />

        <!-- Creating TransactionManager Bean, since JDBC we are creating of type
                DataSourceTransactionManager -->
        <bean id="transactionManager"
                class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- MySQL DB DataSource -->
        <bean id="dataSource"
                class="org.springframework.jdbc.datasource.DriverManagerDataSource">

                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/smlcodes" />
                <property name="username" value="root" />
                <property name="password" value="root" />
        </bean>

        <bean id="customerDAO" class="prog.CustomerDAO">
                <property name="dataSource" ref="dataSource"></property>
```

```
        </bean>

<!--    <bean id="customerManager" class="prog.CustomerService">
            <property name="customerDAO" ref="customerDAO"></property>
        </bean> -->

</beans>
```



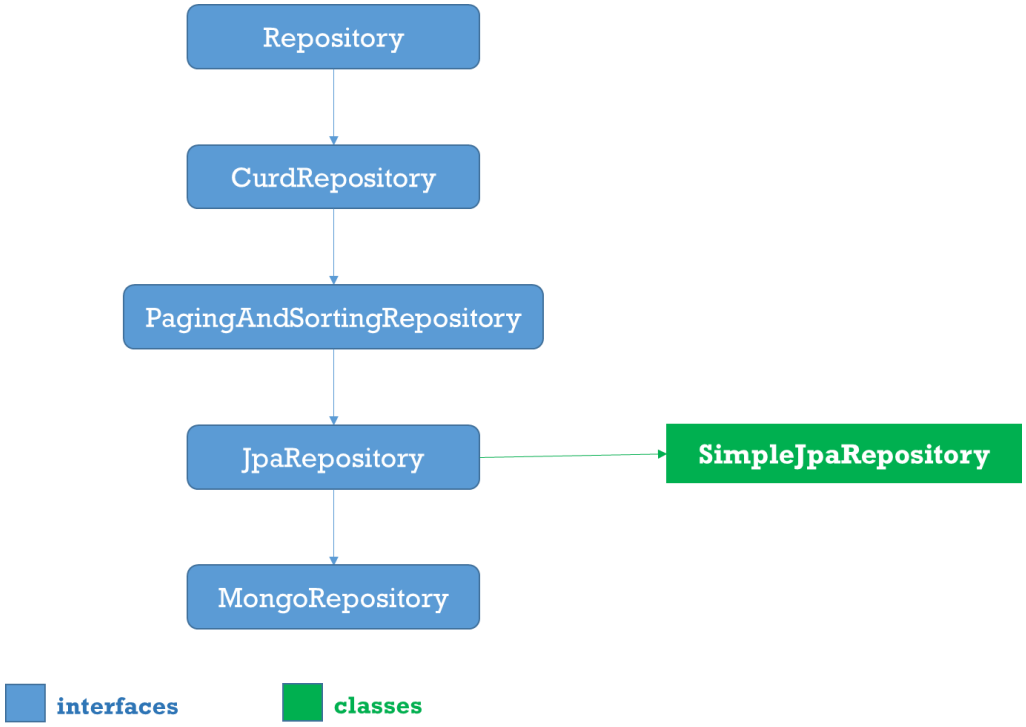
Spring 4 Enhancement – Spring Data Commons

Spring Data Commons provides all the common abstractions that enable you to connect with different data stores.

Spring Data Coomons provides classes & methods, which are common for all the SQL, NoSQL, BigData databases

The Spring Data Commons project provides general infrastructure and interfaces for the other, more specific data projects. Regardless of the type of datastore, Spring Data supports the following aspects with a single API:

- Execute CRUD (create, read, update, delete) operations
- Sorting
- Page-wise reading (pagination)



1.Repositoy

Root interface for all Repositoty classes. It is a **marker interface(no methods)**

2.CurdRepositoy

It provides generic **CRUD** operations irrespective of the underlying database. It extends **Repository** interface.

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
        save(S entity);
        saveAll(Iterable<S> entities);

        Optional<T> findById(ID id);
        Iterable<T> findAll();
        Iterable<T> findAllById(Iterable<ID> ids);

        void deleteById(ID id);
        void delete(T entity);
        void deleteAll(Iterable<? extends T> entities);
        void deleteAll();

        boolean existsById(ID id);
        long count();
}
```

3.PagingAndSortingRepository

PagingAndSortingRepository provides options to

- **Sort** your data using **Sort interface**
- **Paginate** your data using **Pageable interface**, which provides methods for pagination - getPageNumber(), getPageSize(), next(), previousOrFirst() etc

```
public abstract interface PagingAndSortingRepository extends CrudRepository {

    public Iterable findAll(Sort sort);
    public Page findAll(Pageable pageable);

}
```

4.JpaRepository

JPA specific extension of Repository

```
public interface JpaRepository<T, ID extends Serializable> extends
PagingAndSortingRepository<T, ID> {
  List<T> findAll();
  List<T> findAll(Sort sort);
  List<T> save(Iterable<? extends T> entities);
  void flush();
  T saveAndFlush(T entity);
  void deleteInBatch(Iterable<T> entities);
}
```

5.MongoRepository

Mongo specific Repository interface.

```
public interface MongoRepository<T, ID> extends PagingAndSortingRepository {
List<T> findAll()
List<T> findAll(Sort sort)

List<S> saveAll(Iterable<S> entities)
List<S> insert(Iterable<S> entities)
S       insert(S entity)

}
```

6.Custom Repository

- You can create a custom repository extending any of the repository classes - Repository, PagingAndSortingRepository or CrudRepository. For example,

```
interface PersonRepository extends CrudRepository<User, Long> {

}
```

- Spring Data also provides the feature of query creation from interface method names.

```
interface PersonRepository extends Repository<User, Long> {

  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);
```

```
  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

7.Defining Query Methods

The repository proxy has two ways to derive a store-specific query from the method name:

- By deriving the query from the method name directly.

```
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
```

- By using a manually defined query.

```
@Query("select u from User u")
List<User> findAllByCustomQueryAndStream();
```

- Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

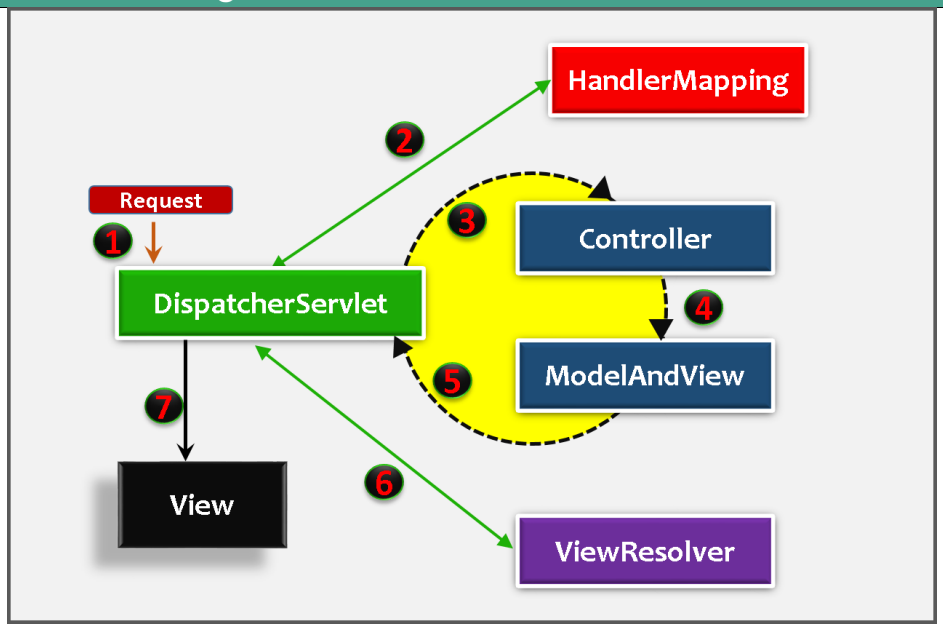
IV. Spring MVC

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general they will consist of POJO.
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.

- The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

Spring MVC Flow Diagram



1. First request will be received by **DispatcherServlet**

2. DispatcherServlet asks **HandlerMapping for Controller class** name for the current request. HandlerMapping will returns controller class name to DispatcherServlet.

3,4,5. DispatcherServlet will transfer request to **Controller**, Controller will process the request by executing appropriate methods and returns **ModeAndView** object. ModeAndView object (contains *Model* data and *View* name) back to the **DispatcherServlet**

6. Now DispatcherServlet send the model object to the **ViewResolver** to get the actual view page
7. Finally, DispatcherServlet will pass the *Model* object to the *View* page to display the result.

Note: All these below Examples are comes under Spring 3.0

Spring MVC –HelloWorld Example

We are going to create following files in this example

1. Create the request, view pages **(index.jsp, hello.jsp)**
2. Create the controller class **(HelloWorldController.java)**
3. Configure entry of controller, Front controller in **web.xml** file
4. Configure ViewResolver, View components in **serveltname-servlet.xml.**
5. Test the Applciation by running on server

```
SpringMVCBasic
    |
    +---src
    |   \---controller
    |           HelloWorldController.java
```

```
    \---WebContent
        |   index.jsp
        \---WEB-INF
            |   hello-servlet.xml
            |   web.xml
            |
            +---jsp
            |       helloView.jsp
            |
            \---lib
```

1.Create the request, view pages (index.jsp, hello.jsp)

```
//index.jsp
<h1>
<a href="helloController">Click Here</a>
</h1>

./jsp/hello.jsp
<h1> SmlCodes :  ${message} </h1>
```

2.Create the controller class –HelloWorldController.java

```
package controller;

@Controller
public class HelloWorldController {
        @RequestMapping("/helloController")
        public ModelAndView helloWorld() {
                String message = "Hello,Spring MVC!!!";
                return new ModelAndView("helloView", "message", message);
                 //it will search for helloView.jsp in JSP Folder
        }
}
```

- To create the controller class, we have to use **@Controller** and **@RequestMapping** annotations.
    - **@Controller** annotation marks this class as **Controller**.
    - **@Requestmapping** annotation is used to map the class with the specified name.
- **@Controller** facilitates **auto-detection** of Controllers which eliminates the need for configuring the Controllers in DispatcherServlte's Configuration file.

- For enabling **auto-detection** of annotated controller's **component-scan** has to be added in configuration file (hello-servlet.xml) **with the package name** where all the controllers are placed.

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

- Controller class returns the instance of **ModelAndView** controller with the mapped name, message name and message value. The message value will be displayed in the jsp page.

3.Configure DispatcherServlet, Controller Entry in web.xml

- In Spring Web MVC, **DispatcherServlet** class works as the front controller. It is responsible to manage the flow of the spring mvc application.
- DispatcherServlet is a normal servlet class which implements **HttpServlet** base class.
- we have to configure DispatcherServlet in **web.xml.**
- Configure <url-pattern>, any url with given pattern will call Spring MVC Front controller.

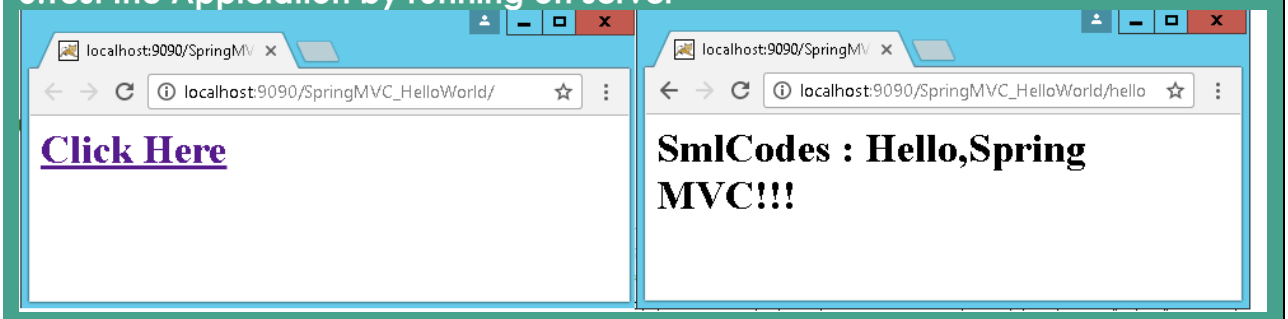
```xml
<!-- file : web.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/</url-pattern> //<url-pattern>*.html</url-pattern>
  </servlet-mapping>
</web-app>
```

4.Configure ViewResolver, View components in hello-servlet.xml

Once the **DispatcherServlet** is initialized, it will looks for a file names **[servlet-name]-servlet.xml** in **WEB-INF** folder. In above example, the framework will look for **hello-servlet.xml**.

```xml
// hello-servlet.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans>
   <context:component-scan base-package="controller"></context:component-scan>

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

</beans>
```

5.Test the Applciation by running on server



Flow of execution

- Run the application, **index.jsp** file will executed, it has a link`<a href="helloController">`
- Once you click on that link, container will check the URL pattern at web.xml and passes the request to the DispatcherServlet
- DispatcherServlet takes this '`helloController`' & asks **HandlerMapping** for **Controller class**
- HaandlerMapping scans the all the controllers classes in the packges & searches for the Class which is conating `helloController` as `@RequestMapping("/helloController")`. Then it will return HelloWorldController class to DispacherServelt.

- DispatcherServlet Excecutes methods which is having `@RequestMapping("/helloController")` in our controller class, that methods gives **ModelAndView** object as return type.

```
return new ModelAndView("helloView", "message", message);
```

- first argument is 'View' page name(`helloView`), second, third are <Key, Value> pair of Model Object for passing data to View Page.

- DispatcherServlet forwards request to *ViewResolver* (hello-servlet.xml) & search for View pages (`helloView.jsp`) location, ViewResolver returns ViewPage to DispatcherServlet

- DispatcherServlet will displays View page with data to the client.

ContextLoaderListener vs DispatcherServlet

In XML based Spring MVC configuration, you must have seen two declarations in `web.xml` file i.e. `ContextLoaderListener` and `DispatcherServlet`. Let's try to understand their purpose in framework and their differences.

Root and child contexts

Before reading further, please understand that –

- Spring can have multiple contexts at a time. One of them will be root context, and all other contexts will be child contexts.
- All child contexts can access the beans defined in root context; but opposite is not true. Root context cannot access child contexts beans.

DispatcherServlet – Child application contexts

`DispatcherServlet` is essentially a Servlet (it extends `HttpServlet`) whose primary purpose is to handle incoming web requests matching the configured URL pattern. It take an incoming URI and find the right combination of controller and view. So it is the front controller.

When you define a `DispatcherServlet` in spring configuration, you provide an XML file with entries of controller classes, views mappings etc. using `contextConfigLocation` attribute.

web.xml

```
<servlet>
    <servlet-name>employee-services</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:employee-services-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

If you do not provide configuration file, then it will load its own configuration file using `[servlet_name]-servlet.xml`. Web applications can define any number

of DispatcherServlet entries. Each servlet will operate in its own namespace, loading its own application context with mappings, handlers, etc.

It means that each DispatcherServlet has access to web application context. Until specified, each **DispatcherServlet** creates own internal web application context.

Starting Spring 3.x, method DispatcherServlet(WebApplicationContext webApplicationContext) create a new DispatcherServlet with the given web application context. It is possible only in Servlet 3.x environment through the ServletContext.addServlet(java.lang.String, java.lang.String) API support.

ContextLoaderListener – Root application context

ContextLoaderListener creates the root application context and will be shared with child contexts created by all **DispatcherServlet** contexts. You can have only one entry of this in web.xml.

web.xml

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/applicationContext.xml</param-value>
</context-param>
```

The context of ContextLoaderListener contains beans that globally visible, like services, repositories, infrastructure beans, etc. After the root application context is created, it's stored in ServletContext as an attribute, the name is:

org/springframework/web/context/ContextLoader.java

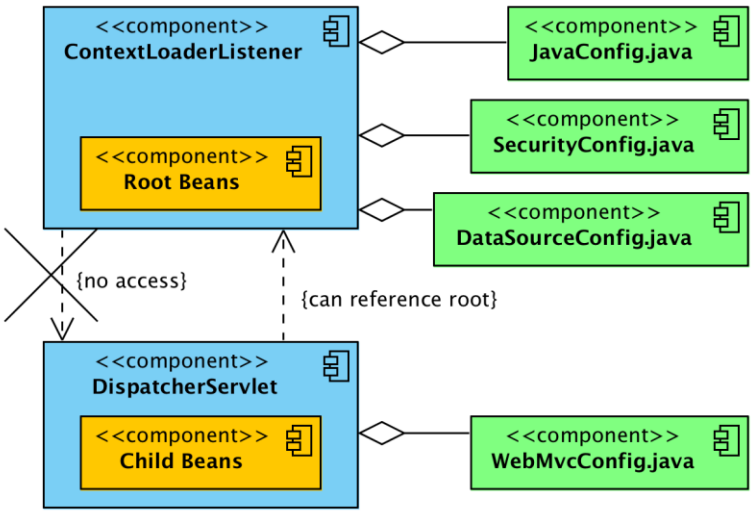
```
servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE,this.context);

//Where attibute is defined in /org/springframework/web/context/WebApplicationContext.java as

WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE = WebApplicationContext.class.getName()
+ ".ROOT";
```

To get root application context in Spring controller, you can use WebApplicationContextUtils class.

Controller.java

```
@Autowired
ServletContext context;

ApplicationContext ac = WebApplicationContextUtils.getWebApplicationContext(context);

if(ac == null){
    return "root application context is null";
}
```

Below image describe the whole relation in single view.



ContextLoaderListener vs DispatcherServlet

1. `ContextLoaderListener` creates root application context.
2. `DispatcherServlet` entries create one child application context per servlet entry.
3. Child contexts can access beans defined in root context.
4. Beans in root context cannot access beans in child contexts (directly).
5. All contexts are added to `ServletContext`.
6. You can access root context using `WebApplicationContextUtils` class.

Generally, you will define all MVC related beans (controller and views etc) in `DispatcherServlet`context, and all cross-cutting beans such as security, transaction, services etc. at root context by `ContextLoaderListener`.

Generally, this setup works fine because rarely you will need to access any MVC bean (from child context) into security related class (from root context). Mostly we use security beans on MVC classes, and they can access it with above setup.

Spring MVC –Multiple Controllers

- We can have many controllers in real-time appplications.in this example we will see how to use multiple controllers in our application. In this example we are taking two controllers
  - **FirstController**
  - **SecondController**

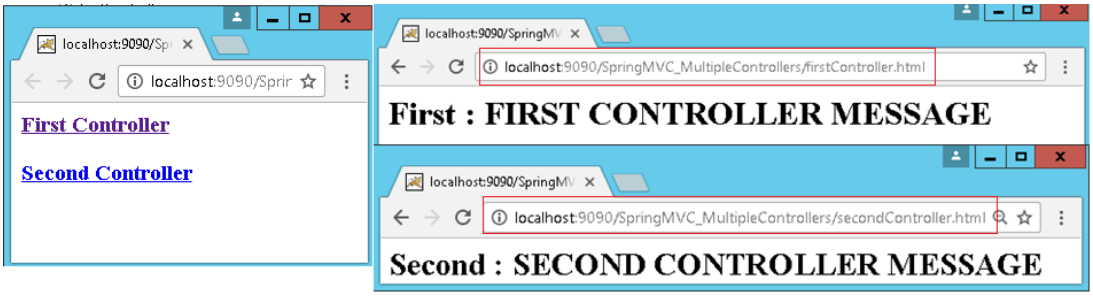
1.View Pages

```
//index.jsp
<a href="firstController.html">First Controller</a>
<a href="secondController.html">Second Controller</a>

//firstView.jsp
<h1> First :  ${m1} </h1>

//secondView.jsp
<h1> Second :  ${m2} </h1>
```

2.Controller Classes

```
//FirstController.java
package controller;
@Controller
public class FirstController {
        @RequestMapping("/firstController")
        public ModelAndView firstMethod(){
                return new ModelAndView("firstView","m1", "FIRST CONTROLLER MESSAGE");
        }
}
//SecondController.java
package controller;
@Controller
public class SecondController {
        @RequestMapping("/secondController")
        public ModelAndView firstMethod(){
                return new ModelAndView("secondView","m2", "SECOND CONTROLLER MESSAGE");
        }
}
```

FrontController configuration **web.xml**, view pages in **hello-servlet.xml** are same as above example



Spring MVC –Request and Response Example

For doing Request & Response type of jobs in Spring MVC, we need to pass **HttpServletRequest** and **HttpServletResponse** objects in the request processing **method of the Controller** class

1.View Pages

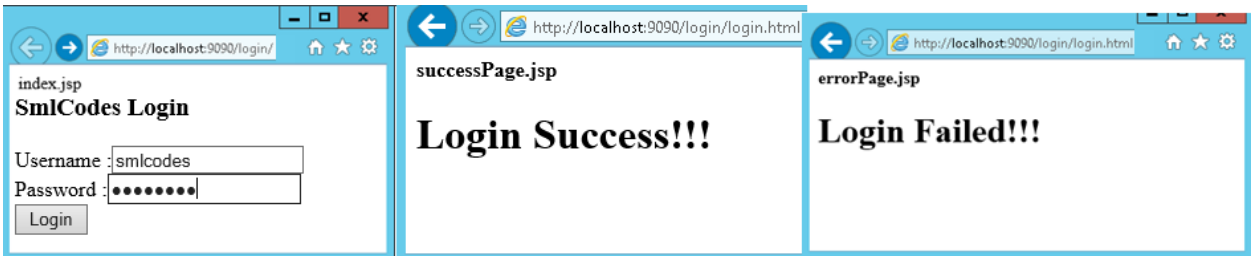
```
//index.jsp
<h3>SmlCodes Login</h3>
<form action="login.html" method="post">
        Username :<input type="text" name="username" /><br />
        Password :<input type="password" name="password" /><br />
        <input type="submit" value="Login" />
</form>

//successPage.jsp
<h1> ${msg}  </h1>
```

```
//errorPage.jsp
<h1> ${msg}  </h1>
```

2.Controller Class: LoginController.java

```
package controller;

@Controller
public class LoginController {

	@RequestMapping("/login")
	public ModelAndView login(HttpServletRequest req, HttpServletResponse res) {
		String username = req.getParameter("username");
		String password = req.getParameter("password");

		if (username.equals(password)) {
			return new ModelAndView("successPage", "msg", "Login Success!!!");
		}
		else {
			return new ModelAndView("errorPage", "msg", "Login Failed!!!");
		}
	}
}
```

FrontController configuration **web.xml**, view pages in **hello-servlet.xml** are same as above example



@RequestMapping

@RequestMapping is one of the most widely used **Spring MVC** annotation. It is used to map web requests onto specific handler classes and/or handler methods.

@RquestMapping annontation can be used in following Levels

1.@RequestMapping –at Class level

If you declare **@RequestMapping** at the class level, the **path will be applicable to all the methods in the class.**

```
@Controller
@RequestMapping(value = "/student")
public class StudentController {
	public ModelAndView addStudent(Student student) {
		return new ModelAndView("addPage", "msg", "Student Added");
	}
}
```

here /**"/student** is enforced to all the methods inside the class. Here we can pass multiple urls to value attribute like

2.@RequestMapping –at Method Level

```
@Controller
@RequestMapping(value = "/student")
public class StudentController {
        @RequestMapping(value = "/add")
        public ModelAndView addStudent(Student student) {
                return new ModelAndView("addPage", "msg", "Student Added");
        }
}
```

Here /add path is applied at method level. To access the addStudent(-) method URL should be ClassURL+MethodUrl = "/student/add

3.@RequestMapping –at HTTP Method Level

Here HTTP methods will filter the handler mappings

```
@Controller
@RequestMapping(value = "/student")
public class StudentController {

    @RequestMapping(value = "/add" method=RequestMethod.GET)
    public ModelAndView addStudent(Student student) {
        return new ModelAndView("addPage", "msg", "Student Added");
    }
    @RequestMapping(value = "/add" method=RequestMethod.POST)
    public ModelAndView addStudent(Student student) {
        return new ModelAndView("addPage", "msg", "Student Added");
    }
}
```

In the above code, if you look at the first two methods mapping to **the same URI, but both have the different HTTP methods**. **First method** will be invoked when HTTP method **GET** is used and the **second** method is invoked when HTTP method **POST** is used.

4.@RequestMapping –Using 'params'

Here the parameters in the query string will filter the handler mappings.

```
@Controller
@RequestMapping(value="/student")
public class HelloWorldController {
@RequestMapping(value="/fetch", params =”sno” )
        public String getSno(@RequestParam("sno") String sno) {
                return "success";
        }
        @RequestMapping(value="/fetch", params = "name")
        public String getName(@RequestParam("name") String name) {
                        return "success";
        }
        @RequestMapping(value="/fetch", params = {"sno=200","name=satya"})
        public String getBoth(@RequestParam("id") String id, @RequestParam("name") String n) {
                        return "success";
        }
}
```

- if request is **/student/fetch? sno=100** then **getSno(-)** will execute.
- if request is **/student/fetch? name=satya** then **getName(-)** will execute.
- if request is **/student/fetch? sno=100&name=satya** then **getBoth(-,-)** will execute.

5. @RequestMapping –Working with Parameters

We have two annotations to process the parameters in given URL. They are

- **@RequestParam**
- **@PathVariable**

@RequestParam

To fetch query string from the URL, @RequestParam is used as an argument.

URL: **/student/fetch? sno=100&name=satya**

```
@Controller
@RequestMapping(value="/student")
public class HelloWorldController {
        @RequestMapping(value="/fetch")
        public String getBoth(@RequestParam("id") String id, @RequestParam("name") String n) {
System.out.println("Sno: "+sno+", Name : "+n)
                        return "success";
        }
}
```

@PathVariable

To access path variable, spring provides **@PathVariable** that is used as an argument. We have to refer the variable in @RequestMapping using **{}**

URL: **/student/fetch/100/satya**

```
@RequestMapping(value="/fetch/{sno}/{name}")
public String getInfo(@PathVariable("sno") String sno, (@PathVariable("sno") String n ) {
System.out.println("Sno:"+sno+", Name : "+n)
        return "success";
}
```

@RequestMapping for Fallback

Using @RequestMapping, we can implement a fallback method. For every response **file not found** exception, this method will be called, in this way we can implement 404 response.

```
@RequestMapping(value="*")
public String default() {
return "success";
}
```

HandlerMapping

When the request is received by **DispatcherServlet, DispatcherServlet asks HandlerMapping for Controller class** name for the current request. HandlerMapping will returns controller class name to DispatcherServlet.

HandlerMapping is an Interface to be implemented by objects that define a mapping between requests and handler objects. By **default**, DispatcherServlet uses **BeanNameUrlHandlerMapping** and **DefaultAnnotationHandlerMapping**. In Spring we majorly use the below handler mappings

- **BeanNameUrlHandlerMapping**
- **ControllerClassNameHandlerMapping**
- **SimpleUrlHandlerMapping**

1.BeanNameUrlHandlerMapping

BeanNameUrlHandlerMapping is the default handler mapping mechanism, which maps **URL requests to the name of the beans**

```
//File : hello-servlet.xml
<beans>
        <bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

        <bean name="/add.htm" class="controller.AddController" />
        <bean name="/update.htm" class="controller.UpdateController" />
        <bean name="/get*.htm" class="controller.GetController" />
</beans>
```

In above example, If URI pattern

- **/add.htm** is requested, DispatcherServlet will forward the request to "`AddController`".
- **/update.htm** is requested, DispatcherServlet will forward the request to "`UpdateController`".
- **/getOneStudent.htm** or **/get{any thing}.htm** is requested, DispatcherServlet will forward the request to the "`GetController`"

2. ControllerClassNameHandlerMapping

ControllerClassNameHandlerMapping use convention to map requested URL to Controller (convention over configuration). It takes the Class name, remove the 'Controller' suffix if exists and return the remaining text, lower-cased and with a leading "/".

By default, Spring MVC is using the `BeanNameUrlHandlerMapping` handler mapping. To enable the **ControllerClassNameHandlerMapping**, declared it in the bean configuration file, and now **the controller's bean's name is no longer required**

```
//File : hello-servlet.xml
<beans>
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />

        <bean class="controller.WelcomeController" />
        <bean class="controller.HelloGuestController" />
</beans>
```

Now, Spring MVC is mapping the requested URL by following conventions :

```
WelcomeController -> /welcome*
HelloGuestController -> /helloguest*
```

- /welcome.htm –> WelcomeController.
- /welcomeHome.htm –> WelcomeController.
- /helloguest.htm –> HelloGuestController.
- /helloguest12345.htm –> HelloGuestController.

- /helloGuest.htm, failed to map **/helloguest***, the "g" case is not match.

To solve the case sensitive issue stated above, declared the "**caseSensitive**" property and set it to true.

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" >
       <property name="caseSensitive" value="true" />
  </bean>
```

3. SimpleUrlHandlerMapping

SimpleUrlHandlerMapping is the most flexible handler mapping class, which allow developer to specify the mapping of URL pattern and handlers explicitly

The property keys are the URL patterns while the property values are the handler IDs or names.

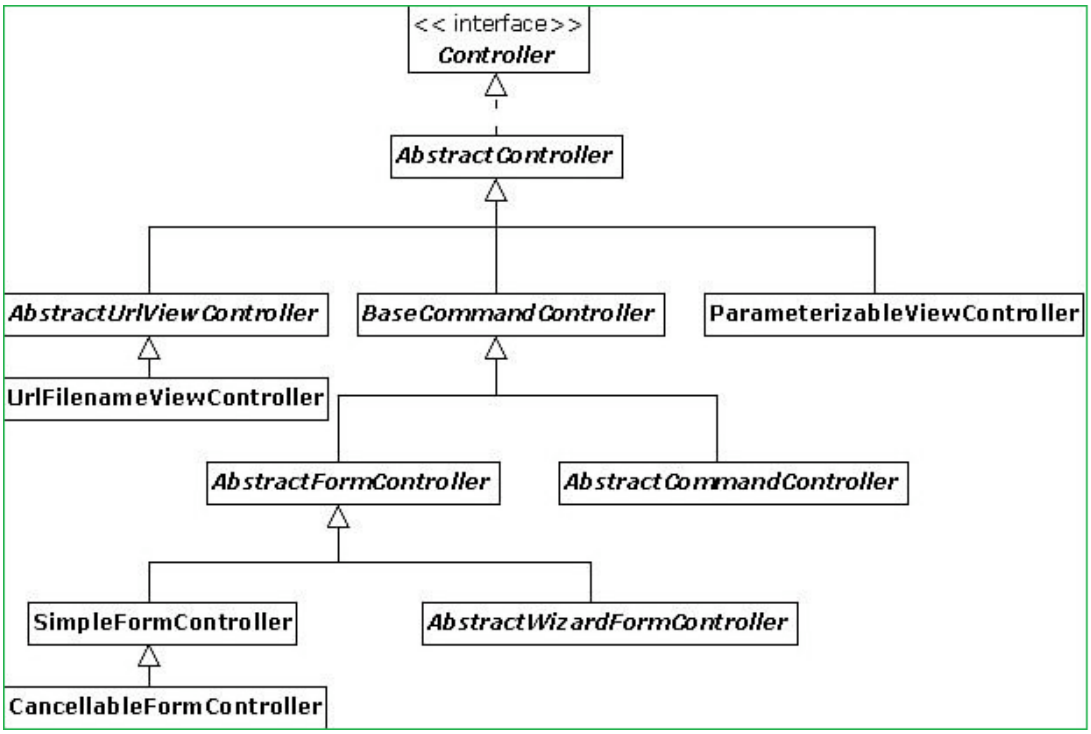
```
<beans>
        <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
                <property name="mappings">
                        <props>
                                <prop key="/welcome.htm">welcomeController</prop>
                                <prop key="/*/welcome.htm">welcomeController</prop>
                                <prop key="/helloGuest.htm">helloGuestController</prop>
                        </props>
                </property>
        </bean>
        <bean id="welcomeController" class="controller.WelcomeController" />
        <bean id="helloGuestController" class="controller.HelloGuestController" />
</beans>
```

- /welcome.htm –> welcomeController.
- /{anything}/welcome.htm –> welcomeController.
- /helloGuest.htm –> helloGuestController.

Controller classes

In this Spring MVC, **DispatcherServlet** works as the Frontcontroller and it delegates the request to the Controller. Developers extends the abstract controller provided by the framework and writes the business logic there. The actual business related processing is done in the Controller.

Spring MVC provides many abstract controllers, which is designed for specific tasks. Here is the list of abstract controllers that comes with the Spring MVC module:



MultiActionController

```
@Deprecated
public class MultiActionController extends AbstractController implements LastModified
```

MultiActionController is used to group related actions into a single controller, the method handler has to follow below signature

```
public (ModelAndView | Map | String | void) actionName(
                HttpServletRequest, HttpServletResponse [,HttpSession] [,CommandObject]);
```

Example: StudentController.java

```
package controller;

public class StudentController extends MultiActionController {
        public ModelAndView add(HttpServletRequest request, HttpServletResponse response) throws Exception
{
                return new ModelAndView("StudentPage", "msg", "addStudent() method");
        }

        public ModelAndView update(HttpServletRequest request, HttpServletResponse response) throws
Exception {
                return new ModelAndView("StudentPage", "msg", "updateStudent() method");
        }

        public ModelAndView delete(HttpServletRequest request, HttpServletResponse response) throws
Exception {
                return new ModelAndView("StudentPage", "msg", "deleteStudent() method");
        }
```

```
        public ModelAndView list(HttpServletRequest request, HttpServletResponse response) throws
Exception {
                return new ModelAndView("StudentPage", "msg", "listStudent() method");
        }
}
```

hello-servlet.xml

```
<beans>

<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

        <bean class="controller.StudentController">
                <property name="methodNameResolver">
                        <bean
        class="org.springframework.web.servlet.mvc.multiaction.InternalPathMethodNameResolver">
                                <property name="prefix" value="check" />
                                <property name="suffix" value="Student" />
                        </bean>
                </property>
        </bean>

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

</beans>
```

Now, the reuqested URL will map to the method name in the following patterns

- **Student**Controller –> **/student/***
- /student/**add**.htm –> **add()**
- /student/**delete**.htm –> **delete()**
- /student/**update**.htm –> **update()**
- /student/**list**.htm –> **list()**

ViewResolvers

In Spring MVC or any web application, for good practice, it's always recommended to put the entire views or JSP files under "**WEB-INF**" folder, to protect it from direct access via manual entered URL.

Those views under **"WEB-INF" folder are named as internal resource views**, as it's only accessible by the servlet or Spring's controllers class.

We have many ViewResolver classes in Spring MVC. Below are the some of those

- **InternalResourceViewResolver**
- **XmlViewResolver**
- **ResourceBundleViewResolver**

1.InternalResourceViewResolver

InternalResourceViewResolver is used to resolve "internal resource view" (in simple, it's final output, jsp or htmp page) **based on a predefined URL pattern**. In additional, it allows you to add some predefined prefix or suffix to the view **name (prefix + view name + suffix),** and generate the final view page URL

1.A controller class to return a view, named "**WelcomePage**".

```
public class WelcomeController extends AbstractController{
        @Override
        protected ModelAndView handleRequestInternal(HttpServletRequest request,
                HttpServletResponse response) throws Exception {
                ModelAndView model = new ModelAndView("WelcomePage");
                return model;
        }
}
```

2. Register **InternalResourceViewResolver** bean in the Spring's bean configuration file.

```
<beans>
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />
        <!-- Register the bean -->
        <bean class="controller.WelcomeController" />

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

Now, Spring will resolve the view's name "**WelcomePage**" in the following way:
prefix + view name + suffix = /WEB-INF/pages/WelcomPage.jsp

Similarly, we have **XmlViewResolver & ResourceBundleViewResolver**, both have their own way of resolving the Views.

Form Handling

Spring framework provides the form specific tags for designing a form. You can also use the simple html form tag also for designing the form. To use the form tag in your JSP page you need to import the Tag Library into your page as.

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

The main difference between HTML tags & Spring form tags is just append **<form: {element}>.**
Examples:`<form:form>, <form:input>, <form:password> <form:radiobutton> etc.,`

In this example we will see the Spring forms and data binding to a controller. Also, we will have a look at ***@ModelAttribute*** annotation

@ModelAttribute

By using *@ModelAttribute* You can map your form fields to a Model class object.

```
@RequestMapping(value = "/addEmployee", method = RequestMethod.POST)
    public String submit( @ModelAttribute("employee") Employee employee ) {
          ----
           ----
        }
```

In above example form data is mapped to **employee** object

SpringMvc –FormHandling Example

View Pages -LoginForm.jsp, LoginSuccess.jsp

```
// LoginForm.jsp
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<style>
.error {
        color: red;
        font-weight: bold;
}
</style>
<form:form action="login" commandName="userForm">
        Email: <form:input path="email" size="30" /><br>
        <form:errors path="email" cssClass="error" /><br>
        Password: <form:password path="password" size="30" /><br>
        <form:errors path="password" cssClass="error" /><br>
        <input type="submit" value="Login" /><br>
</form:form>
</body>
</html>

// LoginSuccess.jsp
<h2>Welcome ${userForm.email}! You have logged in successfully.</h2>
```

Model class –User.java

```
public class User {
        @NotEmpty
        @Email
        private String email;

        @NotEmpty(message = "Please enter your password.")
        @Size(min = 6, max = 15, message = "Your password must between 6 and 15 characters")
        private String password;

        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        public String getPassword() {
                return password;
        }
        public void setPassword(String password) {
                this.password = password;
        }
}
```

- Here we declared validations using Annotations. We will need the **validation-api-1.1.0.Final.jar** and **hibernate-validator-5.0.1.Final.jar** files in order to use the Bean Validation API in our Spring MVC application.
- As we can see, the validation constraint annotations used here are: **@NotEmpty, @Email and @Size.**

We **don't specify error messages for the email field here**. Instead, the error messages for the email field will be specified in a properties file in order to demonstrate localization of validation error messages.

```
//messages.properties
NotEmpty.userForm.email=Please enter your e-mail.
Email.userForm.email=Your e-mail is incorrect.
```

Controller class – LoginController.java

```
@Controller
public class LoginController {
        @RequestMapping(value = "/login", method = RequestMethod.GET)
        public String viewLogin(Map<String, Object> model) {
                User user = new User();
                model.put("userForm", user);
                return "LoginForm";
        }
@RequestMapping(value = "/login", method = RequestMethod.POST)
public String doLogin(@Valid @ModelAttribute("userForm") User userForm, BindingResult result,
                        Map<String, Object> model) {
                if (result.hasErrors()) {
                        return "LoginForm";
                }
                return "LoginSuccess";
        }
}
```

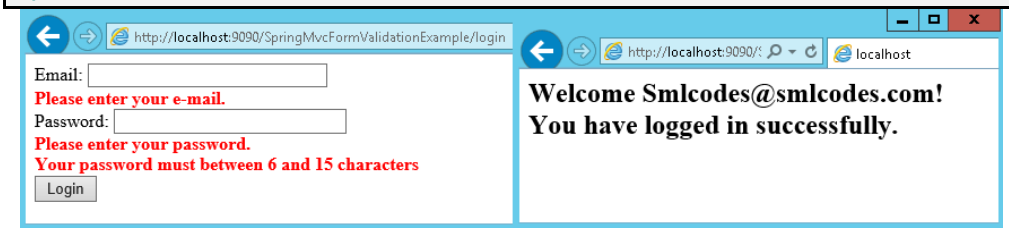
Web.xml

```
<web-app>
        <display-name>SpringMvcFormValidationExample</display-name>
        <servlet>
                <servlet-name>SpringController</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>/WEB-INF/spring-mvc.xml</param-value>
                </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>SpringController</servlet-name>
                <url-pattern>/</url-pattern>
        </servlet-mapping>
</web-app>
```

spring-mvc.xml

```
<beans>
        <mvc:annotation-driven />       //Enable Validation annotations
        <context:component-scan base-package="controller" />

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

        <bean id="messageSource"
        class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
```

```
        <property name="basename" value="/WEB-INF/messages" />
    </bean>
</beans>
```



Themes

Themes in an application can be define as overall **look-and-feel**. Basically, theme is a collection of **static resources like images, CSS etc.** For using theme in your application, you must use interface ***org.springframework.ui.context.ThemeSource***. ***ThemeSource*** is extended by the ***WebApplicationContext*** interface

Buut real work is done by the implementation of **org.springframework.ui.context.support.ResourceBundleThemeSource** that loads properties files from the root of the classpath.

Using ResourceBundleThemeSource, you can define a theme in properties file. You need to make a list of resources inside property file. Given below a sample :

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

The keys of the property file represent the themed element of view. For example : in JSP, you can use `<spring:theme>` custom tag to refer a themed elements. Given below the sample code :

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
<head>
<link rel="stylesheet" href="<spring:theme code='styleSheet'/>" type="text/css"/>
</head>
<body style="background=<spring:theme code='background'/>">
...
</body>
</html>
```

the properties files are loaded from the root of the classpath.

Theme resolvers

After defining theme, you decide which theme to use. The *DispatcherServlet* look for a bean named *themeResolver*to determine which implementation of *ThemeResolver* to use. It detects the theme for a specific request and can also modify the theme of the request.

Spring have following theme resolvers:

Class	Description
FixedThemeResolver	This theme resolver picks fixed theme which can be set using *defaultThemeName* property.

SessionThemeResolver	This theme resolver is used to set the theme for a whole session but not for different session.
CookieThemeResolver	This theme resolver set the selected theme in a cookie for each client.
ThemeChangeInterceptor	This theme resolver changes theme on every request having a simple request parameter

Spring 4 MVC REST Service Example

Spring 4 **@RestController** annotation is introduced. And also we have **@RequestBody, @ResponseBody, @ResponseEntity** annotations which are used to bind the HTTP request/response body with a domain object in method parameter or return type.

@RequestBody

If a method parameter is annotated with **@RequestBody**, Spring will bind the incoming HTTP request body to the method parameter. While doing that, Spring will use HTTP **Message converters** to convert the HTTP request body into class object based on **Accept** header present in request.

- The **Accept** header is used by HTTP clients [browsers] to tell the server what content types they will accept.
- The server sends back the response, which will include a **Content-Type** header telling the client what the content type of the returned content actually is. In case of POST or PUT request, browsers do send data in request, so they actually send content-type as well.

```
@RequestMapping(value="/user/create", method=RequestMethod.POST)
public ResponseEntity<Student> createUser(@RequestBody User user, UriComponentsBuilder ub){
                System.out.println("Creating User "+user.getName());

                if(userService.isUserExist(user)){
                        System.out.println("A User with name "+user.getName()+" already exist");
                        return new ResponseEntity<Void>(HttpStatus.CONFLICT);
                }

                userService.saveUser(user);

                HttpHeaders headers = new HttpHeaders();
            headers.setLocation(ub.path("/user/{id}").buildAndExpand(user.getId()).toUri());
            return new ResponseEntity<Student>(headers, HttpStatus.CREATED);
        }
```

See above, Method parameter **user** is marked with **@RequestBody** annotation

@ResponseEntity

It represents the **entire HTTP response**. Here we can specify status code, headers, and body.

@ResponseBody

If a method is annotated with **@ResponseBody**, Spring will bind the **return value to outgoing HTTP response body**.

While doing that, Spring will use HTTP Message converters to convert the return value to HTTP response body, based on **Content-Type** present in request HTTP header

Spring 4 MVC REST Controller Example

The demo REST application will have Student resource. This student resource can be accessed using standard GET, POST, PUT, DELETE http methods. We will create below REST endpoints for this project.

REST Endpoint	HTTP Method	Description
/students	GET	Returns the list of students
/students/{id}	GET	Returns student detail for given student {id}
/students	POST	Creates new student from the post data
/students/{id}	PUT	Replace the details for given student {id}
/students/{id}	DELETE	Delete the student for given student {id}

1.Set Annotation based Configuration for Spring 4 MVC REST

For this Spring 4 MVC REST tutorial we are going to use Spring's Java based configuration or **annotation based configuration** instead of old XML configuration. So now let us add the Java Configuration required to bootstrap Spring 4 MVC REST in our webapp.

Create `AppConfig.java` file under `/src` folder. Give appropriate package name to your file. We are using `@EnableWebMvc`, `@ComponentScan` and `@Configuration` annotations. These will bootstrap the spring mvc application and set package to scan controllers and resources.

```
package smlcodes.config;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "smlcodes")
public class AppConfig {

}
```

2.Set Servlet 3 Java Configuration

Create **AppInitializer** class under config package. This class will replace **web.xml** and it will map the spring's dispatcher servlet and bootstrap it.

```
package smlcodes.config;
public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

	@Override
	protected Class[] getRootConfigClasses() {
		return new Class[] { AppConfig.class };
	}
	@Override
	protected Class[] getServletConfigClasses() {
		return null;
```

```
	}
	@Override
	protected String[] getServletMappings() {
		return new String[] { "/" };
	}
}
```

3.Create the Student Model

Next let us create `Student` model class that will have few properties such as firstName, lastName, email etc. This bean will hold student information

```
package smlcodes.model;
public class Student {
	private int sno;
	private String name;
	private String address;

	public Student(int sno, String name, String address) {
		super();
		this.sno = sno;
		this.name = name;
		this.address = address;
	}
	public Student() {
		super();
	}
	//Setters & getters
}
```

4.Create the Dummy Student Data Access Object (DAO)

we will create a dummy data access object that will store student details in a list. This DAO class can be easily replaced with Spring Data DAO or custom DAO.

The StudentDAO contains methods `list()`, `get()`, `create()`, `update()` and `delete()` to perform CRUD operation on students.

```
package smlcodes.dao;

@Component
public class StudentDAO {
	private static List<Student> students;

//Instance block
	{
		students = new ArrayList();
		students.add(new Student(101, "Satya", "Hyderabad"));
		students.add(new Student(201, "Vijay", "Banglore"));
		students.add(new Student(301, "Rajesh", "Vijayawada"));
	}

	public List list() {
		return students;
	}

	public Student get(int sno) {
		for (Student c : students) {
			if (c.getSno()==sno) {
				return c;
			}
		}
		return null;
	}

	public Student create(Student student) {
		student.setSno(new Random().nextInt(1000));
		students.add(student);
```

```
                return student;
        }
        public int delete(int sno) {
                for (Student c : students) {
                        if (c.getSno()==sno) {
                                students.remove(c);
                                return sno;
                        }
                }

                return 0;
        }


        public Student update(int sno, Student student) {
                for (Student c : students) {
                        if (c.getSno()==sno) {
                                student.setSno(c.getSno());
                                students.remove(c);
                                students.add(student);
                                return student;
                        }
                }

                return null;
        }
}
```

5.Create the Student REST Controller

Now let us create `StudentRestController` class. This class is annotated with `@RestController`annotation.

Also note that we are using new annotations **@GetMapping, @PostMapping, @PutMapping** and **@DeleteMapping** instead of standard **@RequestMapping**.

These annotations are available since Spring MVC 4.3 and are standard way of defining REST endpoints. They act as wrapper to @RequestMapping. For example @GetMapping is a composed annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.GET).**

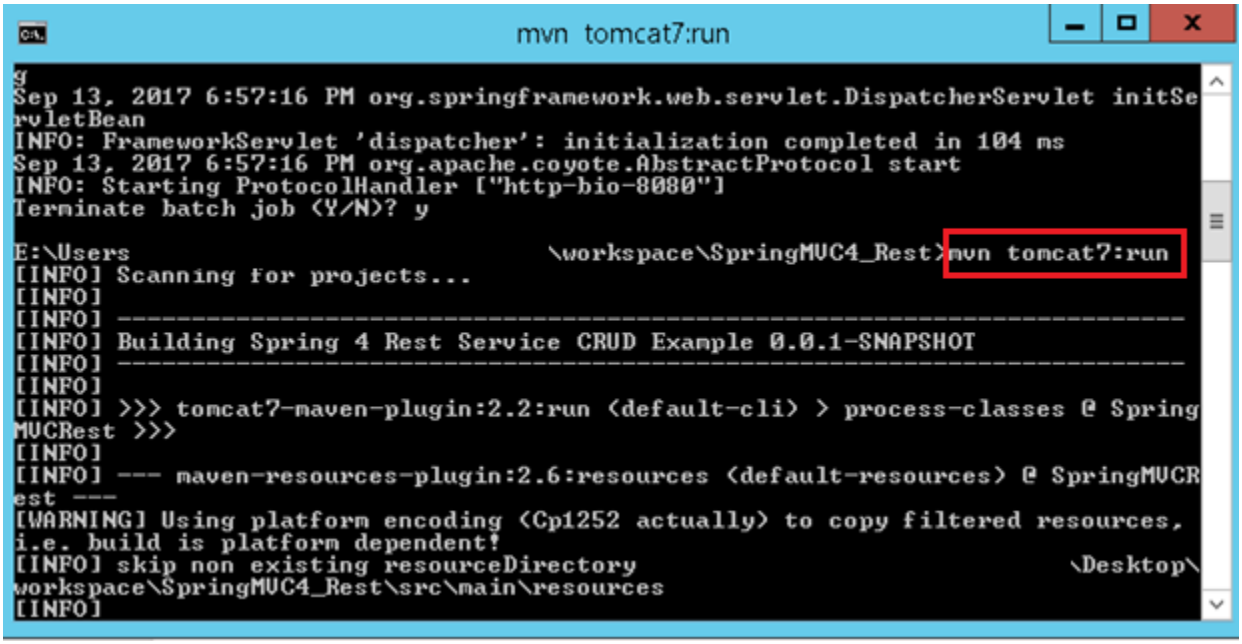
```
package smlcodes.controller;

@RestController
public class StudentRestController {

        @Autowired
        private StudentDAO studentDAO;

        @GetMapping("/students")
        public List getStudents() {
                return studentDAO.list();
        }

        @GetMapping("/students/{sno}")
        public ResponseEntity getStudent(@PathVariable("sno") int sno) {
                Student student = studentDAO.get(sno);
                if (student == null) {
        return new ResponseEntity("No Student found for ID " + sno, HttpStatus.NOT_FOUND);
                }
                return new ResponseEntity(student, HttpStatus.OK);
        }

        @PostMapping(value = "/students")
        public ResponseEntity createStudent(@RequestBody Student student) {
                studentDAO.create(student);
```

```
                return new ResponseEntity(student, HttpStatus.OK);
        }

        @DeleteMapping("/students/{sno}")
        public ResponseEntity deleteStudent(@PathVariable int sno) {
                if (studentDAO.delete(sno) == 0) {
        return new ResponseEntity("No Student found for ID " + sno, HttpStatus.NOT_FOUND);
                }
                return new ResponseEntity(sno, HttpStatus.OK);
        }

        @PutMapping("/students/{sno}")
        public ResponseEntity updateStudent(@PathVariable int sno, @RequestBody Student student) {
                student = studentDAO.update(sno, student);
                if (null == student) {
        return new ResponseEntity("No Student found for ID " + sno, HttpStatus.NOT_FOUND);
                }
                return new ResponseEntity(student, HttpStatus.OK);
        }
}
```

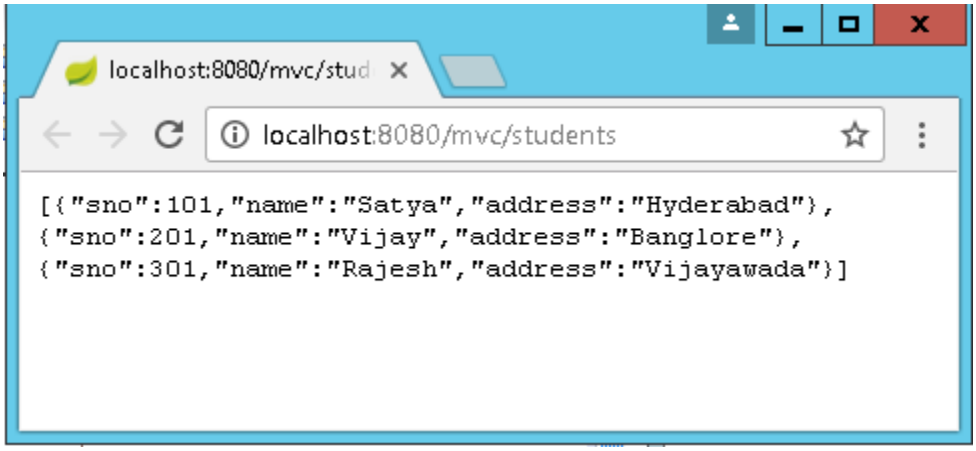
6.Test the Application

To test application, fisrt do **mvn clean install**

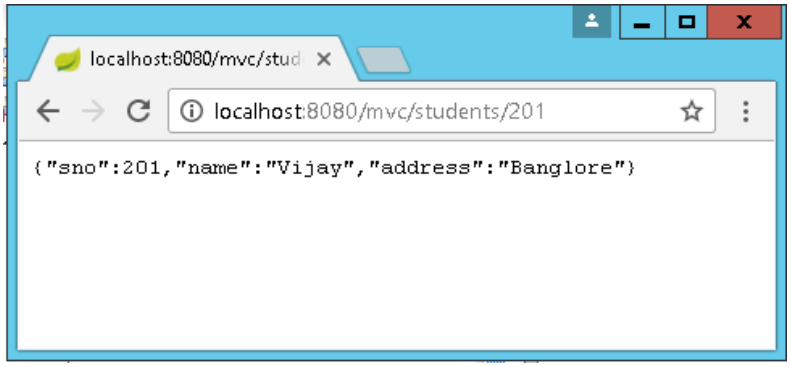
To run application use **mvn tomcat7:run**



All List : http://localhost:8080/mvc/students



Get one : http://localhost:8080/mvc/students/{id}



POST the student details to http://localhost:8080/mvc/students using POSTMan extension

Summary MVC

Spring MVC

```
@Controller
```

```
@RequestMapping("student")
```

```
@RequestMapping(value = "/add" method=RequestMethod.GET)
@RequestMapping(value = "/add" method=RequestMethod.POST)
```

```
/student/fetch/100/satya
@RequestMapping(value="/fetch/{sno}/{name}")
public String getInfo(@PathVariable("sno") String sno, (@PathVariable("sno") String n ) {
}

/student/fetch? sno=100&name=satya
@RequestMapping(value="/fetch")
public String getBoth(@RequestParam("id") String id, @RequestParam("name") String n) {
}

//Form Data
@RequestMapping(value = "/addEmployee", method = RequestMethod.POST)
    public String submit( @ModelAttribute("employee") Employee employee ) {
}
```

WebServices

```
@Path("/student")
```

```
/student/add
@Path("/add")
```

```
@GET
        @Path("/usa")
        @Produces("text/html")

@POST
        @Path("/usa")
        @Produces("text/html")
```

```
@Path("{rollno}/{name}/{address}")
@Produces("text/html")
public Response get(@PathParam("rollno") String rollno,@PathParam("name") String name,
                                        @PathParam("address") String address) {  }


students?rollno=1218&name=SATYA KAVETI&address=VIJAYAWADA
@GET
@Produces("text/html")
public Response get (@QueryParam("rollno") String rollno,@QueryParam("name") String name,
                                        @QueryParam("address") String address) { }

//DefaultValue
@GET
@Produces("text/html")
public Response getResultByPassingValue(@DefaultValue("1000") @QueryParam("rollno") String rollno,
        @DefaultValue("XXXX") @QueryParam("name") String name,
                        @DefaultValue("XXXX") @QueryParam("address") String address) {



customers;custNo=100;custName=Satya
@GET
@Produces("text/html")
public Response getResultByPassingValue(
                        @MatrixParam("rollno") String rollno,
                        @MatrixParam("name") String name,
                        @MatrixParam("address") String address) {}

//Form
@POST
@Path("/registerStudent")
@Produces("text/html")
public Response getResultByPassingValue(
                @FormParam("rollno") String rollno,
                @FormParam("name") String name,
                @FormParam("address") String address) {}


// HeaderParam
@GET
        @Path("/headerparam")
        public Response getHeader(
                 @HeaderParam("user-agent") String userAgent,
                 @HeaderParam("Accept") String accept,
                @HeaderParam("Accept-Encoding") String encoding,
                @HeaderParam("Accept-Language") String lang) {
//Context
@Path("Context ")
        public Response getHttpheaders(@Context HttpHeaders headers){
                 String output = "<h1>@@Context Example - HTTP headers</h1>";
                  output =  output+"<br>ALL headers -- "+ headers.getRequestHeaders().toString();
                  output =  output+"<br>All Cookies -- "+ headers.getCookies().values();
                  return Response.status(200).entity(output).build();
```

Spring 4

```
Sprin3 -@RequestMapping(value="/user/create", method=RequestMethod.POST)
```

```
Spring4 -
@GetMapping("/students/{sno}")
        public ResponseEntity getStudent(@PathVariable("sno") int sno) {
}

@PostMapping(value = "/students")
        public ResponseEntity createStudent(@RequestBody Student student) {
}

@DeleteMapping("/students/{sno}")
        public ResponseEntity deleteStudent(@PathVariable int sno) {
}

public ResponseEntity<Student> createUser(@RequestBody User user, UriComponentsBuilder ub){
}
```

V. Spring JEE

This Module is for implementing the middleware services required for Business logic. This spring JEE module is an abstraction layer on top of RMI, Java mail, JMS, Jars etc.

There is a difference between AOP and JEE modules

- AOP is just for applying the services (or) injecting the services but not for implementing the services, whereas JEE is a module for implementing the services.

- For real time Business logic development with middleware services, we use spring core, spring AOP, and spring JEE modules.

What is Spring Security

Spring security framework focuses on providing both **authentication and authorization** in java applications. It also takes care of most of the common security vulnerabilities such as CSRF attack.

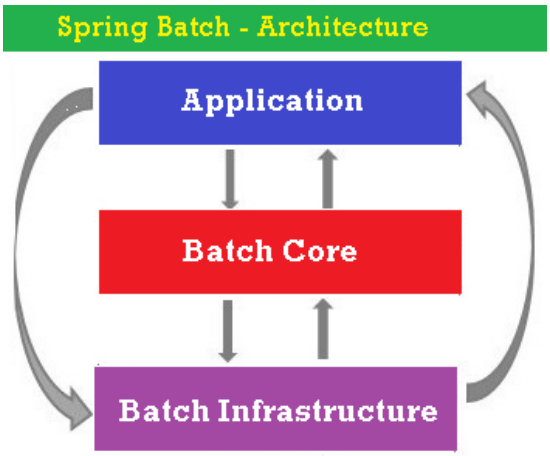
It's very beneficial and easy to use Spring security in web applications, through the use of annotations such as `@EnableWebSecurity`. You should go through following posts to learn how to use Spring Security framework.

- Spring Security in Servlet Web Application
- Spring MVC and Spring Security Integration Example

Spring Batch

In any enterprise application we facing some situations like we want to execute multiple tasks per day in a specific time for particular time period so to handle it manually is very complicated. For handling this type of situation we make some automation type system which execute in particular time without any man power.

Spring Batch provides reusable functions that are essential in processing **large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management.**

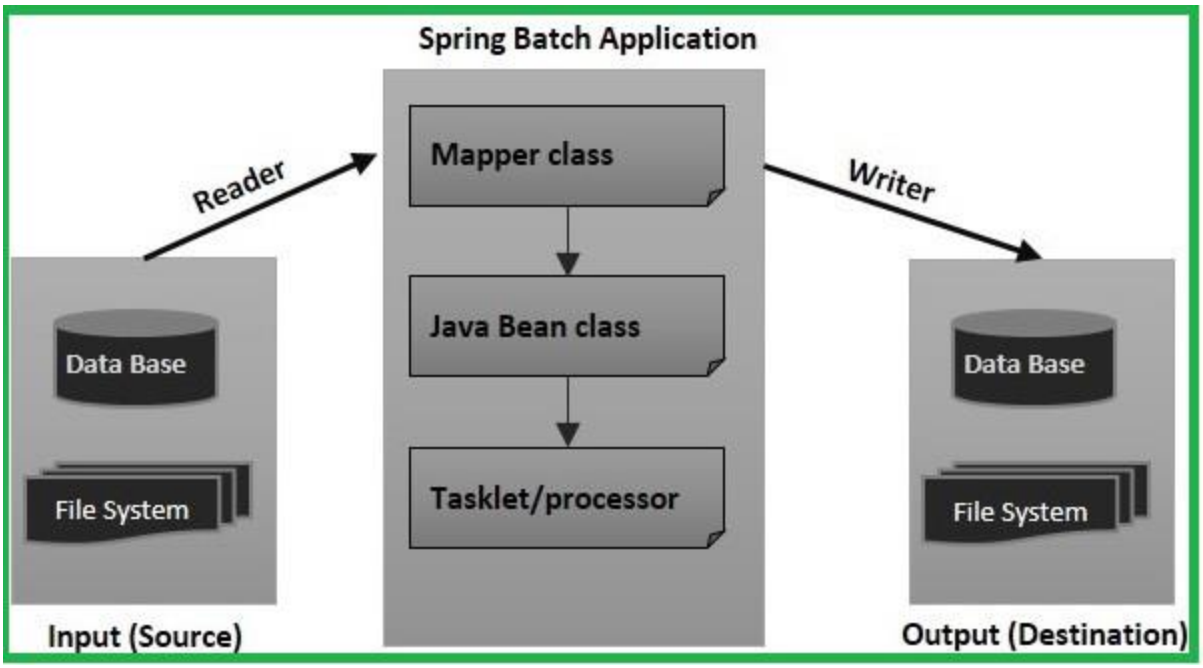


Spring batch contains above 3 components

- **Application** –contains all the jobs and the code we write using the Spring Batch framework.
- **Batch Core** – contains all the API classes that are needed to control and launch a Batch Job.
- **Batch Infrastructure** –contains readers, writers, and services used by both application &Batch components.

Sample SpringBatch Applications

- Spring Batch application read XML data and Writer to MySQL.
- Spring Batch application read CSV data and Writer to XML file.
- Spring Batch application read MySQL data and Writer to XML file.
- Spring Batch application read MySQL data and Writer to TEXT file.



Spring Annotations

Core Spring Annotations

These annotations are used by Spring to guide creation and injection of beans.

ANNOTATION	USE	DESCRIPTION
@Autowired	Constructor, Field, Method	Declares a constructor, field, setter method, or configuration method to be autowired by type. Items annotated with @Autowired do not have to be public.
@Configurable	Type	Used with <context:springconfigured> to declare types whose properties should be injected, even if they are not instantiated by Spring. Typically used to inject the properties of domain objects.
@Order	Type, Method, Field	Defines ordering, as an alternative to implementing the org. springframework.core.Ordered interface.
@Qualifier	Field, Parameter, Type, Annotation Type	Guides autowiring to be performed by means other than by type.
@Required	Method (setters)	Specifies that a particular property must be injected or else the configuration will fail.
@Scope	Type	Specifies the scope of a bean, either singleton, prototype, request, session, or some custom scope.

Example:

```
//File: Student.java
public class Student {
	private int sno;
	private String name;

	@Autowired
	private Address address; //this property is Autowiring
//Setters & getters

	@Autowired
	public Student(Address address) {
		System.out.println("CONSTRCUTOR Injection");
		this.address = address;
	}
	@Autowired
	public void setAddress(Address address) {
		this.address = address;
		System.out.println("Setter Injection");
	}
}
```

- @Autowired annotation is auto wire the bean by matching data type.
- **@Autowired** can be applied on setter method, constructor or a field.in above we applied at 3 places, we need to place at one of the places.

To activiate Spring core annotations in our application, we have to configure *AutowiredAnnotationBeanPostProcessor* bean in SpringConfig.xml

```
<!-- File : SpringConfig.xml -->
<beans>
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor" />
	<bean id="student" class="anno.Student">
		<property name="sno" value="101"></property>
		<property name="name" value="Satya Kaveti"></property>
		<!-- This property will Autowires
		<property name="address">
			<ref bean="addr" />
		</property> -->
	</bean>

	<bean id="address" class="anno.Address">
		<property name="hno" value="322"></property>
		<property name="city" value="HYDERABAD"></property>
	</bean>
</beans>
```

```
//File : AutowireAnnotationExample.java
public class AutowireAnnotationExample {
	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext("anno/SpringConfig.xml");
		Object ob = context.getBean("student");
		Student st = (Student) ob;
		Address addr = st.getAddress();

		System.out.println("Sno : "+st.getSno());
		System.out.println("Name : "+st.getName());
		System.out.println("City : "+addr.getCity());
	}
}
```

```
Sno : 101
Name : Satya Kaveti
City : HYDERABAD
```

The @Qualifier annotation us used to control which bean should be autowire on a field. For example, bean configuration file with two similar person beans.

```
<bean id="address1" class="anno.Address">
                <property name="hno" value="322"></property>
                <property name="city" value="HYDERABAD"></property>
        </bean>
        <bean id="address2" class="anno.Address">
                <property name="hno" value="322"></property>
                <property name="city" value="HYDERABAD"></property>
        </bean>
```

```
public class Student {
        private int sno;
        private String name;
        @Autowired
        @Qualifier("address1")
        private Address address;
}
```

Stereotyping Annotations

ANNOTATION	USE	DESCRIPTION
@Component	Type	Generic stereotype annotation for any Spring-managed component.
@Controller	Type	Stereotypes a component as a Spring MVC controller.
@Repository	Type	Stereotypes a component as a repository. Also indicates that SQLExceptions thrown from the component's methods should be translated into Spring DataAccessExceptions.
@Service	Type	Stereotypes a component as a service.

Spring MVC Annotations

These were introduced in Spring 2.5 to make it easier to create Spring MVC applications with minimal XML configuration and without extending one of the many implementations of the Controller interface.

ANNOTATION	USE	DESCRIPTION
@Controller	Type	Stereotypes a component as a Spring MVC controller.
@InitBinder	Method	Annotates a method that customizes data binding.
@ModelAttribute	Parameter, Method	When applied to a method, used to preload the model with the value returned from the method. When applied to a parameter, binds a model attribute to the parameter. table

ANNOTATION	USE	DESCRIPTION
@RequestMapping	Method, Type	Maps a URL pattern and/or HTTP method to a method or controller type.
@RequestParam	Parameter	Binds a request parameter to a method parameter.
@SessionAttributes	Type	Specifies that a model attribute should be stored in the session.

AOP Annotations

ANNOTATION	USE	DESCRIPTION
@Aspect	Type	Declares a class to be an aspect.
@After	Method	Declares a method to be called after a pointcut completes.
@AfterReturning	Method	Declares a method to be called after a pointcut returns successfully.
@AfterThrowing	Method	Declares a method to be called after a pointcut throws an exception.
@Around	Method	Declares a method that will wrap the pointcut.
@Before	Method	Declares a method to be called before proceeding to the pointcut.
@DeclareParents	Static Field	Declares that matching types should be given new parents,that is, it introduces new functionality into matching types.
@Pointcut	Method	Declares an empty method as a pointcut placeholder method.

Spring Testing

References

- Introductions: http://www.java4s.com/spring/
- IoC,DI : https://www.javatpoint.com/ioc-container
- Setter: java4s.com
- Autowire : http://www.java4s.com/ javatpoint.com, mkyong
- Spring JDBC: javatponit.com
- Spring AOP: mkyong.com, JavaTpoint.com
- SpringEL : http://www.baeldung.com/spring-expression-language
- RequestMapping : http://javabeat.net/requestmapping-spring-mvc/
- Controllers, view resolevrs: mkyong.com
- Form : http://www.codejava.net/
- http://www.beingjavaguys.com/p/spring-framework.html
- https://howtodoinjava.com/java-tutorials-list-howtodoinjava/

References

https://docs.spring.io/spring/docs/5.0.0.RELEASE/spring-framework-reference/core.html#spring-core

1. **Core:** https://docs.spring.io/spring/docs/5.0.0.RC3/spring-framework-reference/core.html#beans
2. **AOP:** https://docs.spring.io/spring/docs/5.0.0.RC3/spring-framework-reference/core.html#aop-introduction
3. **SpringData**: https://docs.spring.io/spring/docs/5.0.0.RC3/spring-framework-reference/data-access.html#jdbc-introduction
4. **SpringWeb**: https://docs.spring.io/spring/docs/5.0.0.RC3/spring-framework-reference/web.html#mvc-introduction
5. **Spring Test**: https://docs.spring.io/spring/docs/5.0.0.RC3/spring-framework-reference/testing.html#unit-testing

Core

Spring Life Cycle : http://javainsimpleway.com/spring-bean-life-cycle/

https://javabeginnerstutorial.com/spring-framework-tutorial/java-spring-bean-lifecycle/

Annotations

https://springframework.guru/spring-framework-annotations/

https://dzone.com/articles/a-guide-to-spring-framework-annotations

Spring Data

https://www.infoq.com/articles/spring-data-intro

http://spring.io/projects/spring-data

http://www.springboottutorial.com/introduction-to-spring-data-with-spring-boot