/smlcodes /smlcodes /smlcodes

Spring

Spring Core, Spring Data, Spring MVC,
Spring AOP, SpringBoot

Small Codes

Programming Simplified

A SmlCodes.Com Small presentation

In Association with Idleposts.com

For more tutorials & Articles visit SmlCodes.com

Spring Tutorial

First published on Sep 2017, Published by **SmlCodes.com**

Author Credits

Name : **Satya Kaveti**

Email : satyakaveti@gmail.com

Website : smlcodes.com

Digital Partners

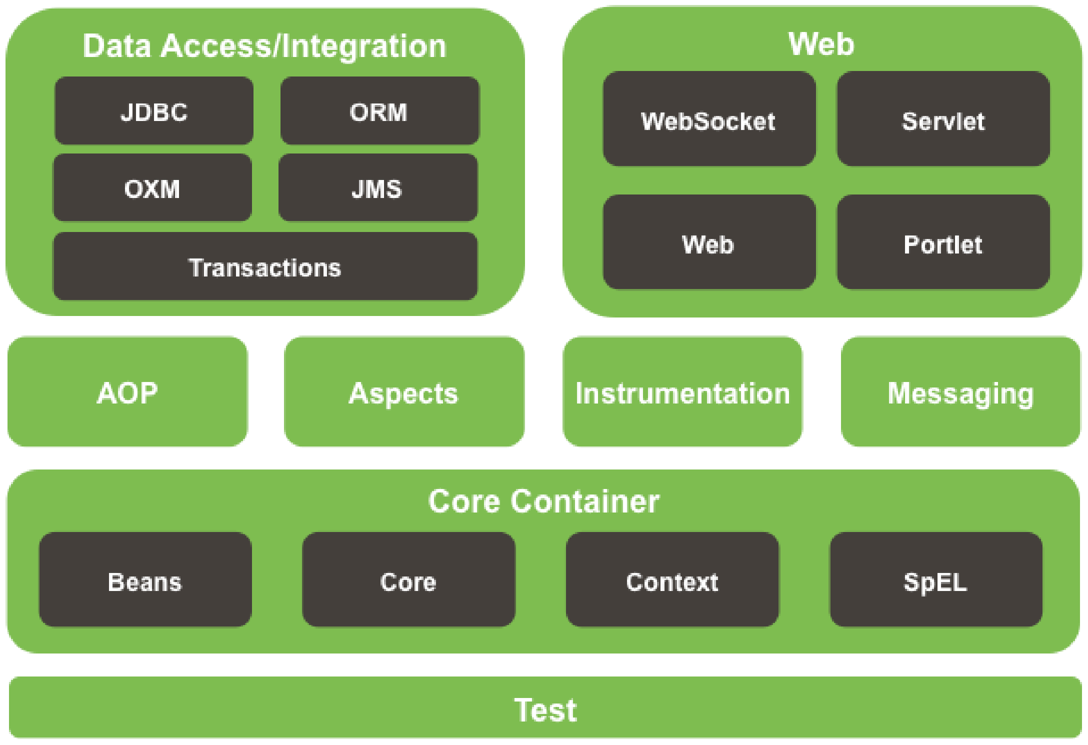
Introduction

Spring is a **light weight, open source and non-invasive** framework created by Rod Johnson in 2003.

- **Spring is light weight framework** because of its **POJO class model**
- **Spring is non-invasive framework**, it doesn't force a programmer to extend or implement their class from any predefined class or interface given by Spring API.But in struts we used to extend Action Class so, struts is invasive framework
- **Simplicity,** Spring is simple because as it is **non-invasive, POJO and POJI model.**
- **Testability,** for writing & testing the spring application, **server is not mandatory**.

Spring Modules

The Spring framework is a layered architecture which consists of several modules. All modules are built on the top of its **core container.** Spring framework is divided into below modules



1. **Spring Core Container:** The Core Container consists of the Core, Beans, Context, and Expression Language modules.
    - The **Core** module provides the fundamental parts of the framework **IOC and DI** features.
    - The **Bean** module provides **BeanFactory**
    - The **Context** module supports **internationalization (I18N), EJB, JMS, Basic Remoting**
    - The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

2. **AOP Module:** These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.

3. **Spring Data (DAO+ORM):** These modules basically provide support to interact with the database, contains **JDBC and Transaction modules**. Spring doesn't provide its own ORM implementation but offers integrations with popular Object Relational mapping tools like Hibernate, JPA, iBATIS SQL Maps etc.

4. **JEE Module:** It also provides support for JMX, JCA, EJB and JMS etc. In lots of cases, JCA (Java EE Connection API) is much like JDBC, except where JDBC is focused on database JCA focus on connecting to legacy systems.

5. **Web Module:** Spring comes with MVC framework which eases the task of developing web applications. It also integrates well with the most popular MVC frameworks like Struts, Tapestry, JSF, Wicket etc.

I. Spring CORE

Tight Coupling Vs. Loose Coupling

Tight Coupling

Tight coupling exists when a group of classes are **highly dependent on one another**.

```
class Car {
	void move() {
		// logic...
	}
}

class Traveler {
	Car c = new Car();
	void startJourney() {
		c.move();
		}
}
```

- In the above example, **Traveler object is depending on car object**. So traveler class creating an object of Car class inside it.
- If method in **car object is changed then we need to do the changes in the Traveler class too.** so it's the **tight coupling between Traveler and Car class objects**

Loose Coupling

A loosely-coupled class can be consumed and tested independently of other (concrete) classes.

Interfaces are a powerful tool to use for decoupling. Classes can communicate through interfaces rather than other concrete classes, and any class can be on the other end of that communication simply by implementing the interface

```
interface Vehicle {
       void move();
       }

class Car implements Vehicle {
       public void move() {
              // logic
       }
}

class Bike implements Vehicle {
       public void move() {
              // logic
       }
}

class Traveler {
       Vehicle v;
       public void setV(Vehicle v) {
              this.v = v;
       }
       void startJourney() {
              v.move();
       }
}
```

In Above **Example if Car object is replaced with Bike then no changes are required in Traveler class**, this means **there is loose coupling between Traveler and Vehicle object.**

Dependency Injection in Java (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**

- The **org. springframework.beans.factory.BeanFactory** and the **org.springframework.context. ApplicationContext** interfaces acts as the IOC container.
- The **ApplicationContext** interface is the **child** interface **of** BeanFactory interface.
- ApplicationContext adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory

1. BeanFactory

- **XmlBeanFactory** is the implementation class
- It only supports **Bean instantiation/wiring**

Using BeanFcatory in our Application

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

2. ApplicationContext

- **ClassPathXmlApplicationContext** class is the implementation class
- It Supports Bean **instantiation/wiring, Automatic BeanPostProcessor registration, Automatic BeanFactoryPostProcessor registration, Convenient MessageSource access (for i18n), ApplicationEvent publication**

Using ApplicationContext in our Application

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

Dependency Injection in Spring (using IoC)

In IOC, if our class object need to get any primitive values or it need to access any other class objects or it may need any collection type to access then, some external person will provide the required things (primitive values or objects or collections) to our class object. **Who is that?**

Here the external person who will provide [Injects] all the required things to our spring bean class is the **Spring IoC Container.**

For injecting the required things to the current spring class (spring bean), spring IoC container will do this in 2 ways by using

1. **Setter injection**
2. **Construction injection**

Spring container knows whether to perform setter or constructor injection by reading the information from an external file called **spring configuration file.**

Before going to further topics, lets install/configure Spring Environment to run the applications.

Spring Installation & Configuration

- Spring is initially released by **interface21**, and now it is renamed as **springsource.org**
- The main jar file of spring is called **spring.jar** it depends on **commons-logging.jar**
- So in order to work with spring framework we need to set following two jars in classpath
    - **spring.jar**
    - **commons-logging.jar**
- Spring is distributed as a **COMBINED** jar file for all modules and also individual jar file for each module [1 for core, 1 for Context, etc.]

Direct Download

- Go to https://repo.spring.io/release/org/springframework/spring/
- Select the release & Download
- Add the required jar's to Classpath.

Maven Installation

Place flowing line of code in **pom.xml** & run the mvn

```
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring</artifactId>
                        <version>2.5.6</version>
                </dependency>

        </dependencies>
</project>
```

Steps to develop any Spring Application

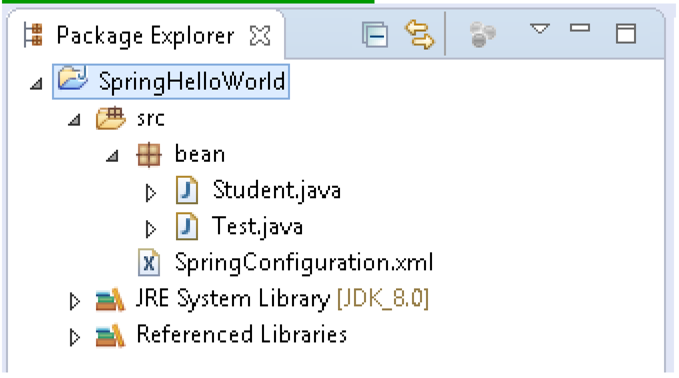
To create any Spring based application, we need follow below steps

1.Create the **Spring Bean class/POJO Class with Setter& getter methods**

2.Create the **SpringConfiguration.xml** & configure **<bean>** with values

3.Create **a Test class, to test our Application**

Example: Hello, World



1.Create the Spring Bean class/POJO Class

```
public class Student {
      private String name;
      public String getName() {
            return name;
      }
      public void setName(String name) {
            this.name = name;
      }
      public void displayInfo() {
            System.out.println("Hello: " + name);
      }
}
```

2. Create the SpringConfiguration.xml file to provide the values

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

      <bean id="ob" class="bean.Student">
            <property name="name" value="Satya Kaveti"></property>
      </bean>
</beans>
```

Configuring SpringConfiguration.xml, with **<bean id="" class="">** and setters as <property name="name" **value="Satya">**

3. Create a Test class, to test our Application

```
public class Test {
public static void main(String[] args) {
      Resource resource=new ClassPathResource("SpringConfiguration.xml");
      BeanFactory factory=new XmlBeanFactory(resource);

      Student student=(Student)factory.getBean("ob");
      student.displayInfo();
}
}
```

```
Hello: Satya Kaveti
```

Test Class Explanation

1.Starting Spring Enviromnet

- Spring environment starts by loading **spring-configuration.xml** file into **Resource Interface** object, is known as bootstrapping of spring framework.
- **This configuration file contains all bean configurations about our application.**
- **Resource is an interface** and **ClassPathResource is an implemented class** given by spring, both are given in org.springframework.core.io. *

```
Resource resource=new ClassPathResource("SpringConfiguration.xml");
```

2.Loading IoC into our Application

- The IoC Container will load into our application, when while reading **spring-configuration.xml**, through **Resource object**.
- **Spring IOC container is called BeanFactory** and this container is responsible for creating the bean objects and for injecting its dependencies throughout our applications.
- **BeanFactory is an interface** and **XmlBeanFactory is an implementation class** of it, BeanFactory given in org.springframework.beans.factory. * and XmlBeanFactory is given in org.springframework.beans.factory.xml.* pack

```
BeanFactory factory=new XmlBeanFactory(resource);
```

3.get Bean Objects

- Now you can get required object from spring container by calling **getBean()** method, while calling this method we need to pass the bean id as a parameter like. getBean(bean id), and this method always returns Object class object and we need to type caste this into our bean type.

```
Object ob = factory.getBean("st");
Student s1 = (Student)ob;
```

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

- Primitive Types
- Object Types
- Collection Types

1.Setter Injection with Primitive Types

In this example we will see the Setter injection with primitive values like int, String, float, etc.,

1.Create the Spring Bean class: Student.java

```java
package bean;
public class Student {
	private int sno;
	private String name;
	private String address;

	//Setters & Getters
}
```

2.Create the SpringConfig.xml & configure <bean> with values

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

	<bean id="ob" class="bean.Student">
		<property name="sno" value="101"></property>
		<property name="name" value="Satya Kaveti"></property>
		<property name="address" value="HYDERABAD"></property>
	</bean>
</beans>
```

3.Create a Test class, to test our Application : SetterInjectionExample.java

```java
package bean;
public class SetterInjectionExample {
public static void main(String[] args) {
	Resource resource = new ClassPathResource("SpringConfig.xml");
	BeanFactory factory = new XmlBeanFactory(resource);

	Object ob = factory.getBean("ob");
	Student st = (Student) ob;

	System.out.println(st.getSno());
	System.out.println(st.getName());
	System.out.println(st.getAddress());
}
}
```

```
log4j:WARN Please initialize the log4j system properly.
101
Satya Kaveti
HYDERABAD
```

Explanation

- In **Student.java**, we have 3 properties, spring container will inject values for those properties at run time
- In **Springconfig.xml, <property />** means we are saying to the spring container that we have written setter method in our bean, in the that property we assigned value as an attribute, which means the setter injection is in the form of primitive values [may be int, string, float, etc.,]
- In **SetterInjectionExample.java** first we created `Resource resource` Object by load the configuration file, and gave `resource` object to with XmlBeanFactory, so now **factory** knows all the beans in the xml file so we can now call any bean with bean id.
- if we call **getBean("ob")** then internally the spring framework executes the following statements

```java
		Student ob = new Student();
		ob.setSno(100);
		ob.setName("Satya");
		ob.setAddress("HYDERABAD");
```

By default, every spring bean is the singleton class (Only one object per class). Spring IOC container makes a spring bean as singleton automatically.

2.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 **SpringConfiguration.xml**, we have 2 ways to inform to the spring container about this object dependency
  - Using **<ref />** element
  - Using Inner beans

<ref> Tag

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

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).

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 parenet=”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)**

SpringConfig1.xml

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

SpringConfig2.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**

1.Create the Spring Bean class : Student.java, Address.java

```
package obj;

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

```
package obj;

public class Address {
	private int hno;
	private String city;
//Setters & getters
}
```

2.Create the Spring Configuration xml: StudentConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
	<bean id="st" class="obj.Student">
		<property name="sno" value="101"></property>
		<property name="name" value="Satya Kaveti"></property>
		<property name="address">
			<ref local="adr" />
		</property>
	</bean>

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

3.Create a Test class, to test our Application:

```
package obj;

public class SetterObjectApp {
  public static void main(String[] args) {
	Resource resource = new ClassPathResource("obj/StudentConfig.xml");
	BeanFactory factory = new XmlBeanFactory(resource);

	Student student = (Student) factory.getBean("st");
	Address address = student.getAddress();
	System.out.println(student.getSno() + "," + student.getName());
	System.out.println(address.getHno() + "," + address.getCity());
   }
}
```

```
101,Satya Kaveti
305,HYDERABAD
```

3.Setter Injection with Collections

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.

<list >

- **List allows Duplicate Values**
- If in our spring bean has a property of type **List** then in the **SpringConfig.xml** file, we need to use **<list>** element to inform the Spring IOC container regarding this property
- In SpringConfig.xml, we can use **<value /> and <ref />** tags as sub elements of <set> tag
- 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>
```

<set >

- **Set Doesn't allow Duplicate Values**
- In SpringConfig.xml, we can use **<value /> and <ref />** tags as sub elements of <set> tag

- 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>
```

<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>
```

Example: Setter Injection with Collections –Primitive Types

```
//File: Country.java
public class Country {
	private String countryName;
	private List<String> states; //Allow Duplicates
	private Set<String> rivers; //Not-Allow Duplicates
	private Map<String, String> capitals;
	
	//Setters & getters
}
```

```
<!-- File : SpringConfig.xml -->
	<bean id="ob" class="collections.Country">
		<property name="countryName" value="INDIA"></property>
		<property name="states">
			<list>
				<value>ANDHRA</value>
				<value>ANDHRA</value>
				<value>TELANGANA</value>
				<value>TAMILNADU</value>
			</list>
		</property>
		<property name="rivers">
			<set>
				<value>KRINSHNA</value>
				<value>KRINSHNA</value>
				<value>GANGA</value>
				<value>YAMUNA</value>
			</set>
		</property>
		<property name="capitals">
			<map>
				<entry key="ANDHRA" value="VIJAYAWADA"></entry>
				<entry key="TELANGANA" value="HYDERABAD"></entry>
				<entry key="TAMILNADU" value="CHENNAI"></entry>
			</map>
		</property>
	</bean>
</beans>
```

```
//File : SIMapStringExample.java
public class SIMapStringExample {
	public static void main(String[] args) {
		Resource resource = new ClassPathResource("collections/SpringConfig.xml");
		BeanFactory factory = new XmlBeanFactory(resource);
		
		Object ob = factory.getBean("ob");
		Country c = (Country) ob;
		
		System.out.println("Country Name : " + c.getCountryName());
		System.out.println("States : " + c.getStates());
		System.out.println("Rivers : " + c.getRivers());
		System.out.println("Capitals : " + c.getCapitals());
	}
}
```

```
Country Name : INDIA
States : [ANDHRA, ANDHRA, TELANGANA, TAMILNADU]
Rivers : [KRINSHNA, GANGA, YAMUNA]
Capitals : {ANDHRA=VIJAYAWADA, TELANGANA=HYDERABAD, TAMILNADU=CHENNAI}
```

Example: Setter Injection with Collections –Object Types

```java
//File : Country.java
public class Country {
	private String countryName;
	private List<State> states;
	//Setters & getters
}
```

```java
//File : State.java
public class State {
	private String stName;
	private String stCapital;
	//Setters & getters
}
```

```xml
<!-- File : SpringConfig.xml -->
	<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>
	<bean id="list2" class="collectionsref.State">
		<property name="stName" value="TAMILNADU"></property>
		<property name="stCapital" value="CHENNAI"></property>
	</bean>
</beans>
```

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

		Object ob = factory.getBean("ob");
		Country c = (Country) ob;
		List<State> states = c.getStates();

		System.out.println("Country Name : " + c.getCountryName());
		Iterator<State> itr = states.iterator();
		while (itr.hasNext()) {
			State s = (State) itr.next();
			System.out.println(s.getStName() + " : " + s.getStCapital());
		}
	}
}
```

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

4. 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;
        }
}
```

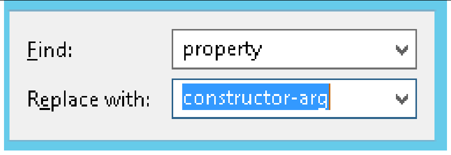
```
<beans>
	<bean id="st" class="obj.Student">
		<constructor-arg name="sno" value="101"></constructor-arg>
		<constructor-arg name="name" value="Satya Kaveti"></constructor-arg>
		<constructor-arg name="address">
			<ref bean="adr" />
		</constructor-arg>
	</bean>
	<bean id="adr" class="obj.Address">
		<constructor-arg name="hno" value="305"></constructor-arg>
		<constructor-arg name="city" value="HYDERABAD"></constructor-arg>
	</bean>
</beans>
```

3. Constrcutor Injection –Colelction 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>
```

If you observe clearly CI is same as SI, just it is replacing <property> tag with <constructor-arg> & remove name attribute



Find: property

Replace with: constructor-arg

Example: Constructor Injection

```java
//File : Country.java
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
//File : State.java
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
<!-- File : SpringConfig.xml -->
<beans>
	<bean id="ob" class="constructorInjection.Country">
		<constructor-arg value="INDIA"></constructor-arg>
		<constructor-arg>
			<list>
				<ref bean="list1" />
				<ref bean="list2" />
			</list>
		</constructor-arg>
	</bean>


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

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

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

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

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

Difference Between Setter Injection & Constructor Injection

Setter Injection	Constructor Injection
1.Partial injection possible: if we have 3 dependencies like int, string, long, then it's 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 within 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

3. Spring Bean Autowiring

In previous Dependency injection we wrote the bean properties explicitly into SpringCongig.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.byName

- 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**.

Example: Autowire –byName

```
//File: Student.java
public class Student {
	private int sno;
	private String name;
	private Address address; //Here ‘address’ is the Bean property name

	//Setters& getteers
}
```

```
//File: Address.java
public class Address {
	private int hno;
	private String city;
	//Setters& getteers
}
```

```
<!-- File : SpringConfig.xml -->
<beans>
       <bean id="student" class="byName.Student" autowire="byName">
              <property name="sno" value="101"></property>
              <property name="name" value="Satya Kaveti"></property>
              <!-- This property will Autowires by name
              <property name="address"> <ref
                     bean="addr" />
              </property> -->
       </bean>

      <bean id="address" class="byName.Address">  //Here it will comapir with bean prop. name
              <property name="hno" value="322"></property>
              <property name="city" value="HYDERABAD"></property>
      </bean>
</beans>
```

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

              Object ob = factory.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("Hno : "+addr.getHno());
              System.out.println("City : "+addr.getCity());
       }
}
```

```
Sno : 101
Name : Satya Kaveti
Hno : 322
City : HYDERABAD
```

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

2.byType

In 'by**Type**" 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**.

Example: Autowire –byType

```
//File: Student.java
public class Student {
       private int sno;
       private String name;
       private Address address; //Here 'address' is 'Address' class type.

       //Setters& getteers
}
```

```java
//File: Address.java
public class Address {
	private int hno;
	private String city;
	//Setters& getteers
}
```

```xml
<beans>

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

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

In above example, Student bean contains Address as a property, Spring container will search in SpringConfig.xml for the bean property whose class type is Address. i.e. byType.Address = address

Note: **if you have multiple bean of one type, it will throw the exception**.

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

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


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

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'student' defined in class path resource
[byType/SpringConfig.xml]: Unsatisfied dependency expressed through bean
property 'address': : No unique bean of type [byType.Address] is defined:
expected single matching bean but found 2: [address, address1]; nested
exception is org.springframework.beans.factory.NoSuchBeanDefinitionException:
No unique bean of type [byType.Address] is defined: expected single matching
bean but found 2: [address, address1]
```

3. constrcutor

- 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**.

Example: Autowire – constructor

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

	public Student(Address address) {
		this.address = address;
	}
	//Setters& getteers for sno, name
}
```

```
//File: Address.java
public class Address {
	private int hno;
	private String city;
	//Setters& getteers
}
```

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

II. Spring Data

Spring Data is a high level project developed by Spring community aimed at simplifying the data access operations for the applications. There are several sub-projects maintained for the individual as follows

- Spring Data Commons
- Spring Data MongoDB
- Spring Data Redis
- Spring Data Solr
- Spring Data Gemfire
- Spring Data REST
- Spring Data Neo4j

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 database 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.

1. JdbcTemplate DDL/DML Operations

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
 ------------
```

2. 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 Data –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");
       }
}
```

Spring Data –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.

```java
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
<?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 Data –@Transactional Annotation

In General, transaction management code needs to be explicitly written so as to commit when everything is successful and rolling back if anything goes wrong. The transaction management code is tightly bound to the business logic in this case

```
       session = HibernateUtil.getSessionFactory().openSession();
       tx = session.beginTransaction();
       tx.setTimeout(5);
       //doSomething(session);
       tx.commit();
```

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 –

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

@Transactional annotation

At a high level, when a class declares **@Transactional** on itself or its members, Spring creates a proxy that implements the same interface(s) as the class you're annotating. In other words, Spring wraps the bean in the proxy and the bean itself has no knowledge of it. A proxy provides a way for Spring to inject behaviors before, after, or around method calls into the object being proxied.

Internally, its the same as using a transaction advice (using AOP), where a proxy is created first and is invoked before/after the target bean's method.

The generated proxy object is supplied with a **TransactionInterceptor**, which is created by Spring. So when the **@Transactional** method is called from client code, the **TransactionInterceptor** gets invoked first from the proxy object, which begins the transaction and eventually invokes the method on the target bean. When the invocation finishes, the **TransactionInterceptor** commits/rolls back the transaction accordingly.

```
public class CustomerManagerImpl implements CustomerManager {
	private CustomerDAO customerDAO;

	public void setCustomerDAO(CustomerDAO customerDAO) {
		this.customerDAO = customerDAO;
	}

	@Override
	@Transactional
	public void createCustomer(Customer cust) {
		customerDAO.create(cust);
	}
}
```

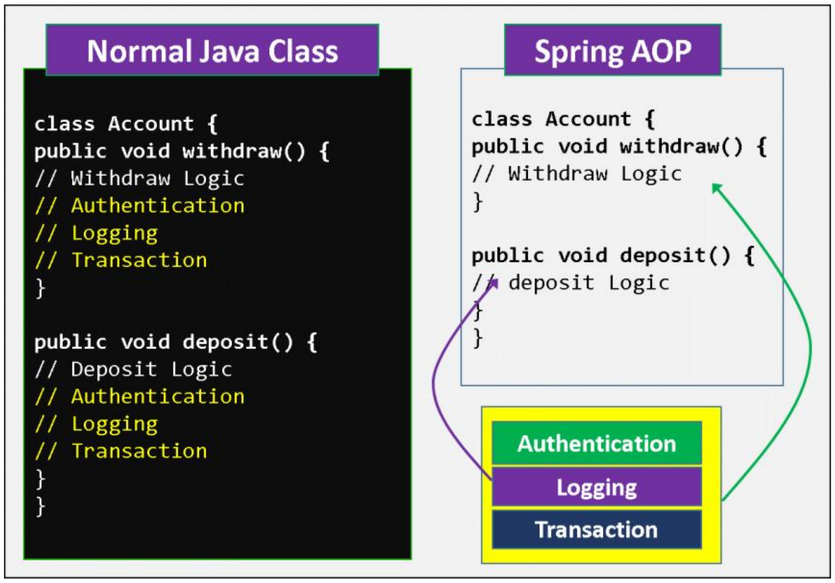
if you notice the CustomerManager implementation, it's just using CustomerDAO implementation to create the customer but provide declarative transaction management through annotating createCustomer() method with @Transactional annotation. That's all we need to do in our code to get the benefits of Spring transaction management.

@Transactional annotation can be applied over methods as well as whole class. If you want all your methods to have transaction management features

III. 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.



```
class Account {
public void withdraw() {
// Withdraw Logic
// Authentication
// Logging
// Transaction
}

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

```
class Account {
public void withdraw() {
// Withdraw Logic
}

public void deposit() {
// deposit Logic
}
}
```

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

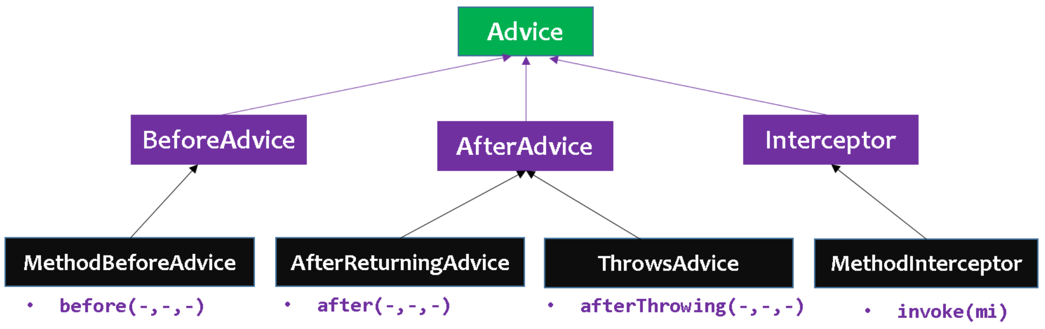
1. **Aspect**
2. **Advice**
3. **JoinPoint**
4. **Pointcut**
5. **Introduction**
6. **Target**
7. **Proxy**
8. **Weaving**
9. **Adviser**

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

Spring AOP –AspectJ

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";
	}
```

```java
        @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.

```xml
<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.

```java
// 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();
        }
}
```

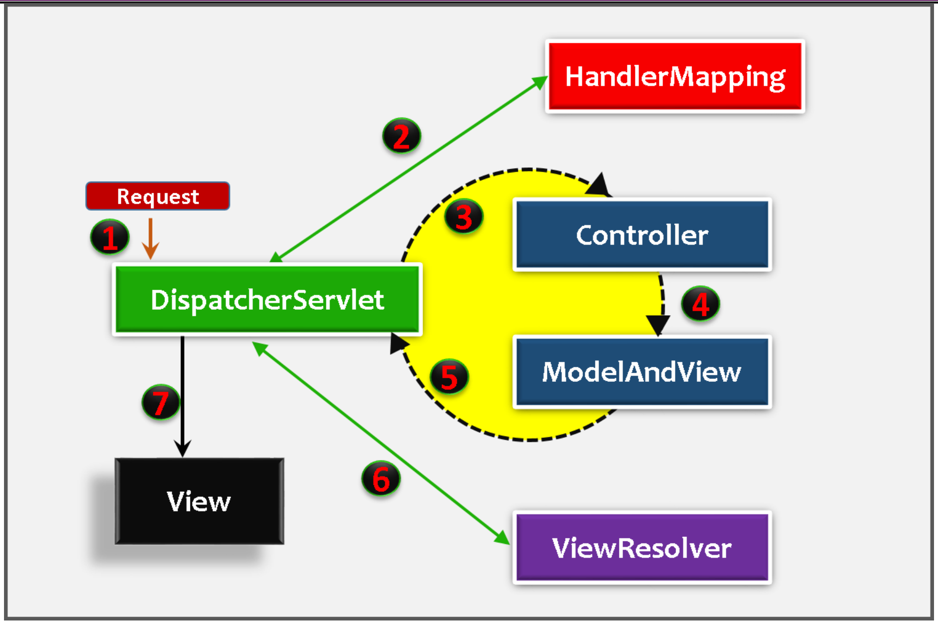
```
Satya : new Student Added
Exception in thread "main" studentAround() is running, args : SATYA
studentThrowException() is running
java.lang.Exception: Student Error
        at StudentImpl.studentThrowException(StudentImpl.java:17)
        at AspectJTestApp.main(AspectJTestApp.java:20)
```

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.

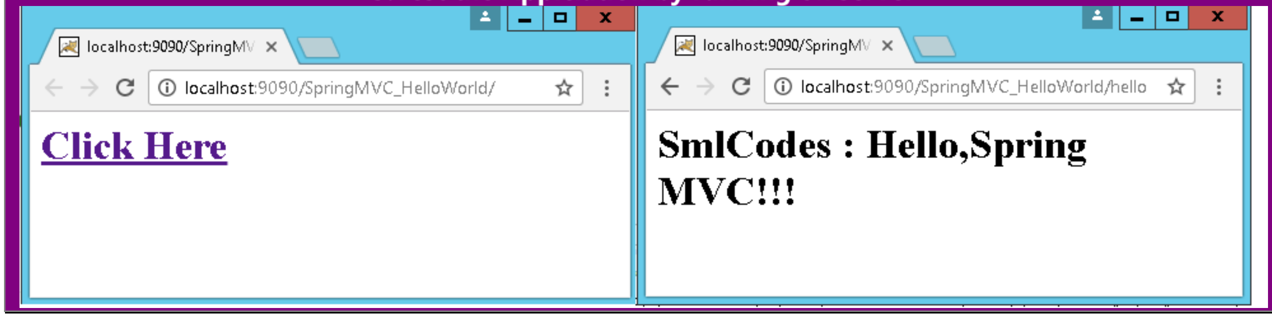
```
<!-- file : web.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/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

```
// 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


localhost:9090/SpringMV
localhost:9090/SpringMVC_HelloWorld/
localhost:9090/SpringMV
localhost:9090/SpringMVC_HelloWorld/hello

Click Here

SmlCodes : Hello,Spring MVC!!!

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 verifies this '*helloController*' name with the string in `@RequestMapping("/helloController")` in our controller class if same it will execute the following method, which 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

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

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

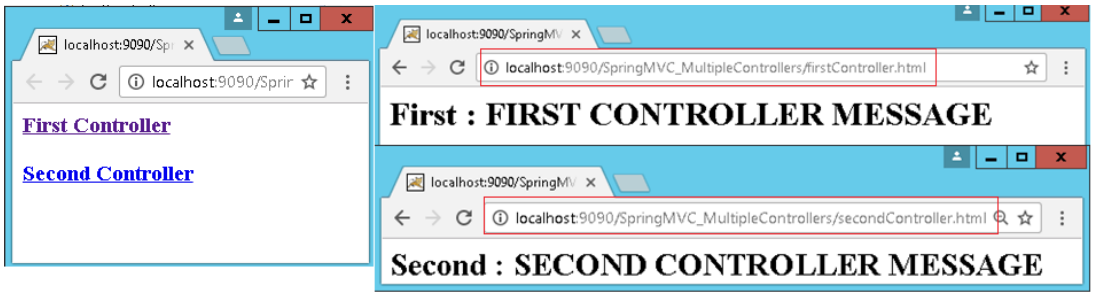
```
//index.jsp
<a href="firstController.html">First Controller</a>
<a href="secondController.html">Second Controller</a>

//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


localhost:9090/Spr

First Controller

Second Controller

localhost:9090/SpringMVC_MultipleControllers/firstController.html

First : FIRST CONTROLLER MESSAGE

localhost:9090/SpringMVC_MultipleControllers/secondController.html

Second : SECOND CONTROLLER MESSAGE

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

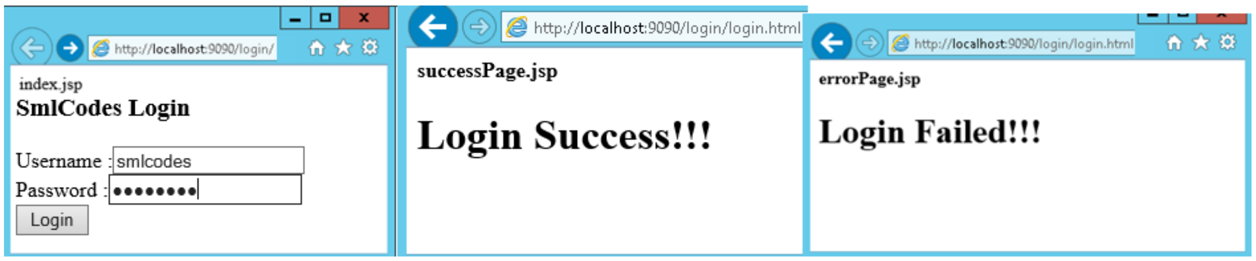
1.View Pages

```
//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


http://localhost:9090/login/

index.jsp

SmlCodes Login

Username : smlcodes

Password : ••••••••

Login

http://localhost:9090/login/login.html

successPage.jsp

Login Success!!!

http://localhost:9090/login/login.html

errorPage.jsp

Login Failed!!!

@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.

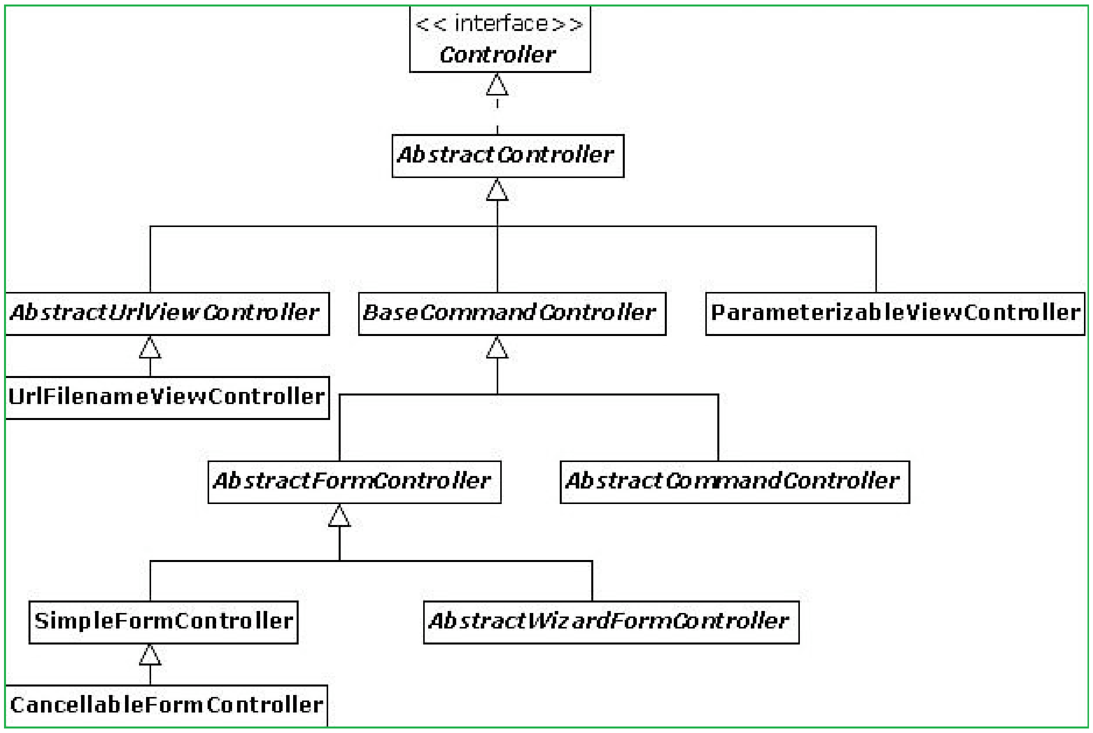
```
<beans>
	<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
		<property name="mappings">
			<props>
				<prop key="/welcome.htm">welcomeController</prop>
				<prop key="/*/welcome.htm">welcomeController</prop>
				<prop key="/helloGuest.htm">helloGuestController</prop>
			</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 controller 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:


<< interface>>
Controller
AbstractController
AbstractUrlViewController
BaseCommandController
ParameterizableViewController
UrlFilenameViewController
AbstractFormController
AbstractCommandController
SimpleFormController
AbstractWizardFormController
CancellableFormController

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

```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

```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

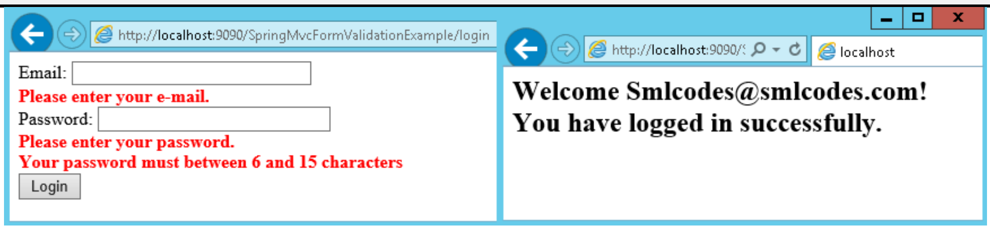
```
@Controller
public class LoginController {
        @RequestMapping(value = "/login", method = RequestMethod.GET)
        public String viewLogin(Map<String, Object> model) {
                User user = new User();
                model.put("userForm", user);
                return "LoginForm";
        }
@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>
```



Email:
Please enter your e-mail.
Password:
Please enter your password.
Your password must between 6 and 15 characters

Welcome Smlcodes@smlcodes.com!
You have logged in successfully.

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;
	{
		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.

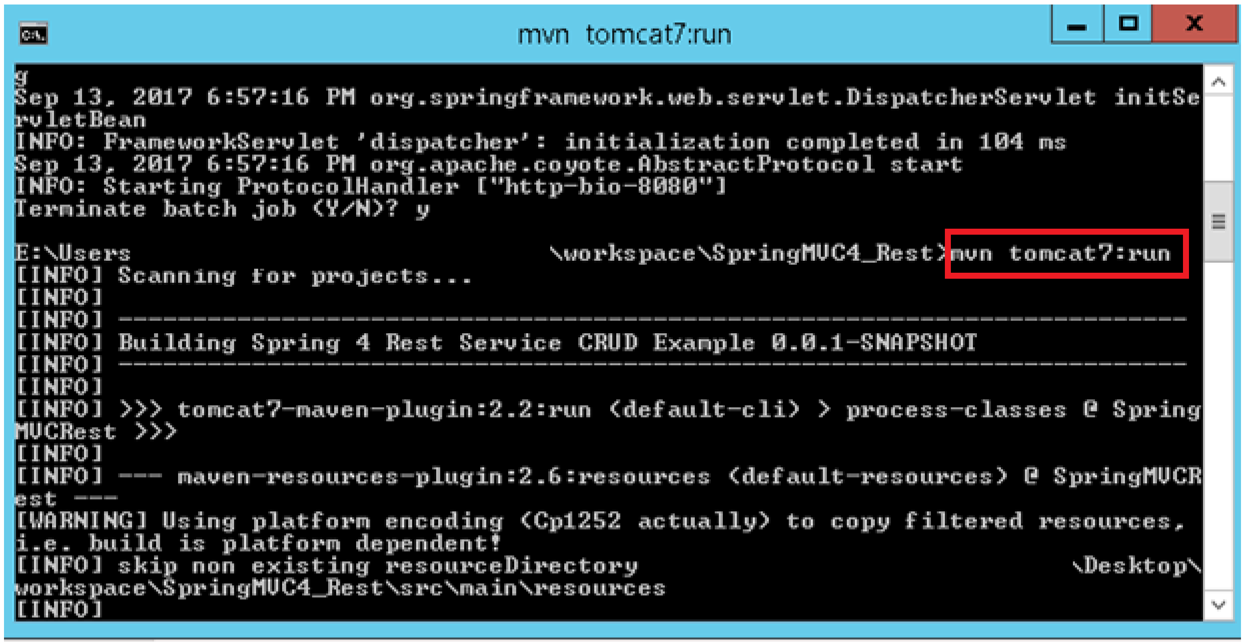
Also note that we are using new annotations @GetMapping, @PostMapping, @PutMapping and @DeleteMapping instead of standard @RequestMapping. These annotations are available since Spring MVC 4.3 and are standard way of defining REST endpoints. They act as wrapper to @RequestMapping. For example @GetMapping 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

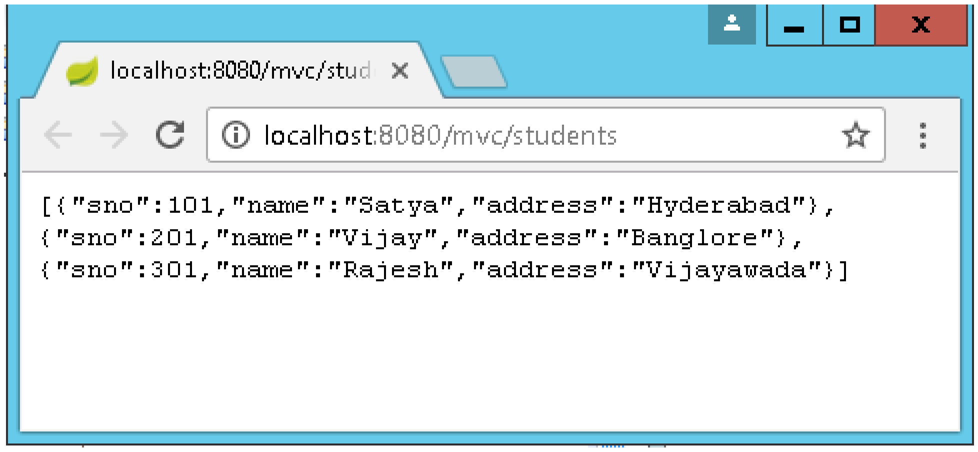
To test application, fisrt do **mvn clean install**

To run application use **mvn tomcat7:run**


mvn tomcat7:run

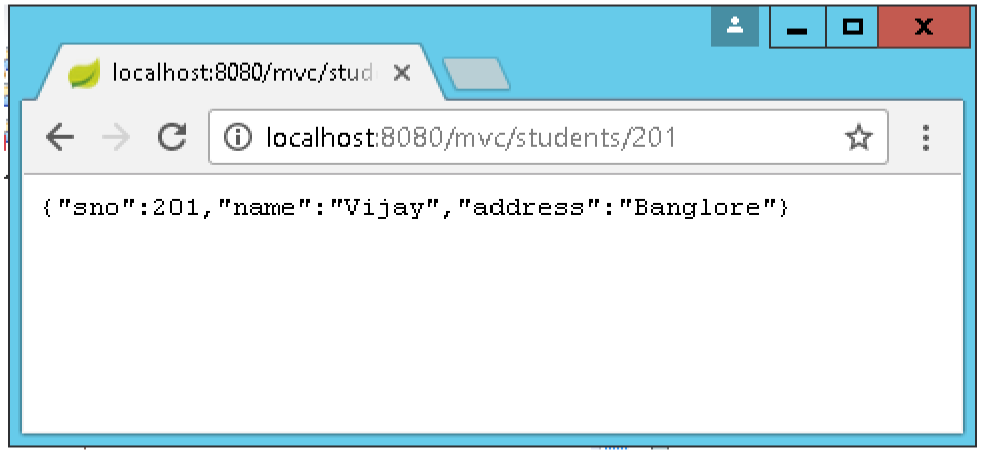
```
g
Sep 13, 2017 6:57:16 PM org.springframework.web.servlet.DispatcherServlet initSe
rvletBean
INFO: FrameworkServlet 'dispatcher': initialization completed in 104 ms
Sep 13, 2017 6:57:16 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
Terminate batch job (Y/N)? y

E:\Users                    \workspace\SpringMVC4_Rest>mvn tomcat7:run
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Spring 4 Rest Service CRUD Example 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> tomcat7-maven-plugin:2.2:run (default-cli) > process-classes @ Spring
MVCRest >>>
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ SpringMVCR
est ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory                          \Desktop\
workspace\SpringMVC4_Rest\src\main\resources
[INFO]
```

All List : http://localhost:8080/mvc/students



```
[{"sno":101,"name":"Satya","address":"Hyderabad"},
{"sno":201,"name":"Vijay","address":"Banglore"},
{"sno":301,"name":"Rajesh","address":"Vijayawada"}]
```

Get one : http://localhost:8080/mvc/students/{id}



```
{"sno":201,"name":"Vijay","address":"Banglore"}
```

POST the student details to http://localhost:8080/mvc/students using POSTMan extension

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.

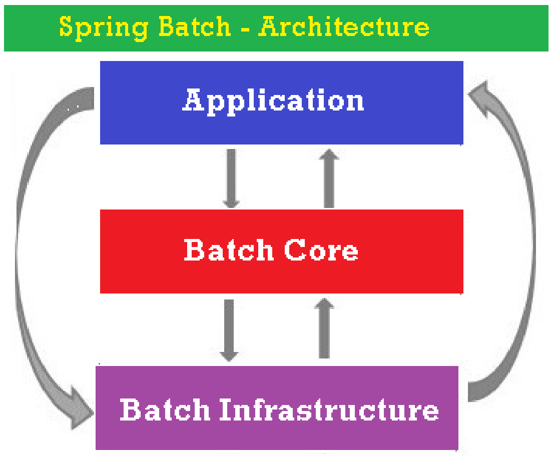
It's very beneficial and easy to use Spring security in web applications, through the use of annotations such as `@EnableWebSecurity`. You should go through following posts to learn how to use Spring Security framework.

- 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.**

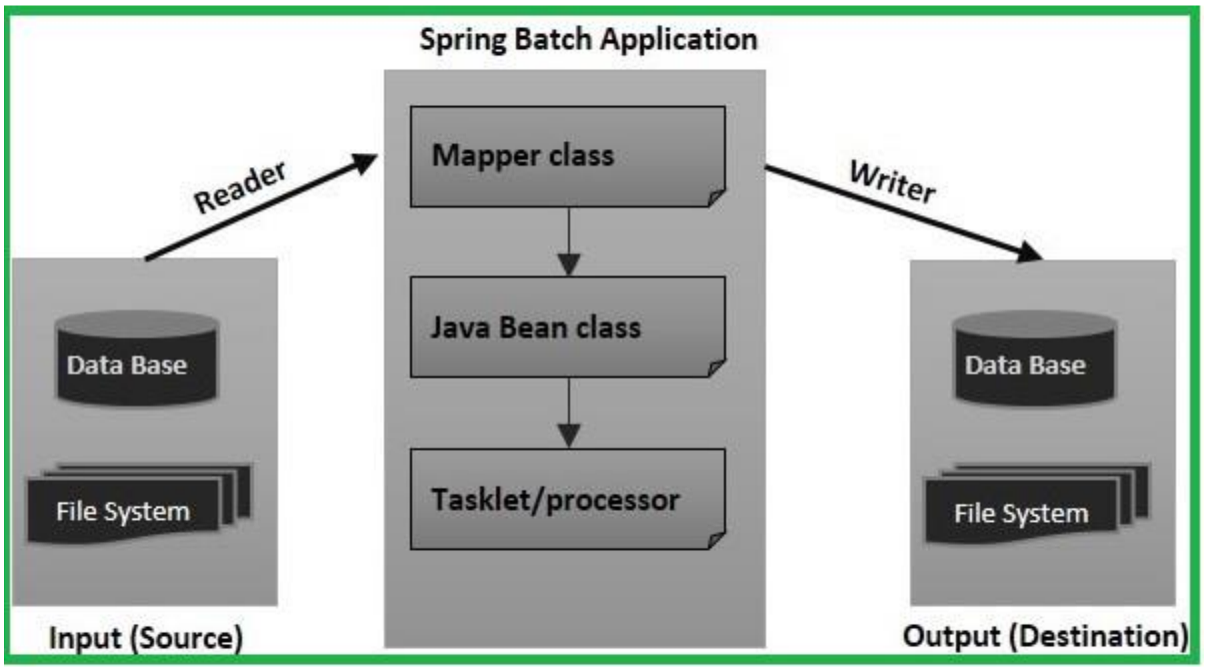


Spring batch contains above 3 components

- **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
@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.

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/

spring
Small Codes