SPRING 4

[Spring Core, Spring AOP, Profiling, Spring-Jdbc, Spring-Hibernate, Spring-Txs, Spring-MVC, Design Patterns, N-Tier Architecture]

K.RAMESH

ASPIRE Technologies

#501, 5th Floor, Mahindra Residency, Maithrivanam Road, Ameerpet, Hyderabad

Ph: 07799 10 8899, 07799 20 8899

E-Mail:ramesh@java2aspire.com website: www.java2aspire.com

1.INTRODUCTION

What is Spring?

Spring is an Integration Framework for developing Enterprise Applications easily. Spring is an open-source framework, developed by Rod Johnson.

Spring's Pledge

J2EE should be Easy to use.

Spring framework simplifies (addresses) complexity of enterprise applications because it uses Java beans to implement enterprise applications that were previously possible only with enterprise beans.

Pre-Requisites

- Java Beans
- Factory Design Pattern
- Template-Callback Pattern

Spring Features

- Dependency Injection (DI) / Inversion Of Control (IOC)
- AOP (Aspect Oriented Programming)
- Lightweight Container

Dependency Injection (DI) / Inversion Of Control (IOC)

All dependent objects are automatically instantiated and injected to bean class instead of creating and looking for dependent objects. Hence, Inversion Of Control is also called as Dependency Injection. The responsibility is transferred away from the bean class and towards its dependencies.

The Advantage with DI is promoting **loose coupling**.

Aspect Oriented Programming (AOP)

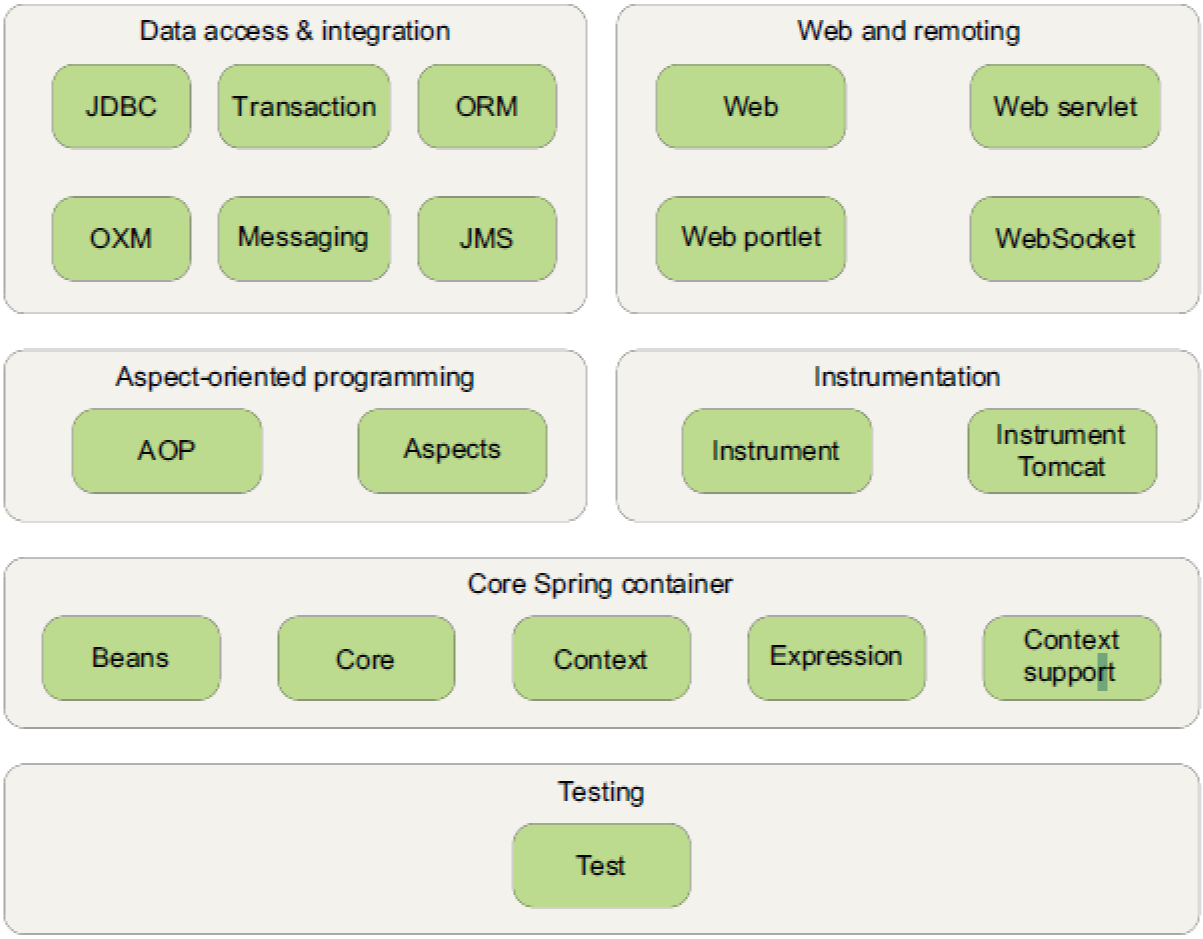
Separating application business logic from system services such as logging, transaction management, persistence, connection pooling, etc.

Lightweight Container

Spring is a lightweight container. The entire spring framework can be distributed in a JAR file. The spring container manages the bean lifecycle. Spring Dependency Injection is also resolved by the container.

Spring Modules

The spring framework consisting of **six** well-defined means main modules. We can selectively choose required modules rather than all for application implementation. The spring modules are built on top of the core container, which defines how beans are created, configured, and managed.



The Spring Framework is made up of six well-defined module categories.

1. **Core Spring container**
   The core container provides the fundamental functionality of the spring framework. The primary component in this module is BeanFactory and ApplicationContext, an implementation of the FactoryPattern with IOC. Also, this module supplies many enterprise services such as email, JNDI, EJB integration, and scheduling.

2. **Spring AOP Module**
   Spring AOP separates business logic from system services such as Logging, Transaction management, Persistence, Application Resources (eg: connection pooling).

3. **Data Access and Integration**

The Template based approach in Spring JDBC API avoids the repetitive code like getting connection, creates a Statement, process ResultSet, and then close the connection.

Spring does not have its own ORM framework, instead it will integrate with several popular ORM frameworks such as Hibernate, iBATIS, TopLink, JDO, OJB, etc. Spring transaction management supports each of these ORM frameworks as well as JDBC.

4. **Web and Remoting**

The web context module builds on the application context module, providing a web context that is required for web based applications. As a result, the spring framework supports third party MVC frameworks like Struts, Webwork, Portlet, Tapestry, etc.

Spring comes with a full-featured Model/View/Controller (MVC) framework for building web applications. Spring's MVC framework uses IoC to provide for a clean separation of controller logic from business objects (POJOs). It also allows us to declaratively bind request parameters to our POJOs. Spring MVC encourages different view technologies like JSPs, Velocity, XSLT, Freemarker, etc.

Spring integrates with several popular MVC frameworks such as STRUTS, JSF, Web Work, and Tapastry.

Spring Remoting capabilities include RMI, Hessian, Burlap, JAX-WS, Spring's own HTTP invoker, REST.

5. **Instrumentation**

6. **Testing**

Spring provides a module to test spring application.

New Features in Spring 2.5

1. Annotation driven development greatly reduces XML-based configuration. Also, supports JSR-250 annotations.
2. Auto-detection of spring component that are annotated with @Component annotation.
3. Annotation driven Spring MVC programming model that greatly simplifies Spring Web development.
4. Full Java 6 and EE 5 support including JDBC 4.0, JAX-WS 2.0, etc.
5. Test framework based on JUnit4 and annotations.

New Features in Spring 3.0

1. Full-scale REST support in Spring MVC, including Spring MVC controllers that respond to REST-style URLs with XML, JSON, etc.
2. New Spring Expression Language (SpEL) is a powerful way of wiring values into a bean's properties or constructor arguments using expressions that are evaluated at runtime.
3. Support for declarative validation with JSR-303 annotations.
4. Java based (using annotations) configuration model that allows for nearly XML-free spring configuration.

New Features in Spring 4.0

1. Spring Boot
2. Spring DATA
3. Leverages Java 8 features such as Lambda expresssions, etc
4. @RestController newly added
5. @GetMapping and @PostMapping, etc
6. @ConditionalOnClass newly added
7. …

Application #1: Spring Hello
This example explains how spring container manages the bean lifecycle and echos greeting message on console.

The following files are required:

1. Interface
2. Implementation bean class
3. Spring Configuration [XML Config / Java Config]
4. Client Code

//GreetingService.java

```
Package edu.aspire.beans;
public interface GreetingService {
        public void sayGreeting();
}
```

// GreetingServiceImpl.java

```
Package edu.aspire.beans;
public class GreetingServiceImpl implements GreetingService {
        private String greeting;
        public GreetingServiceImpl() { } //no-arg constructor
        public GreetingServiceImpl(String greeting) { // Initialize through constructor
                this.greeting = greeting;
        }
        public void setGreeting(String greeting) { // Initialize through setter method.
                this.greeting = greeting;
        }
        public void sayGreeting() {
                System.out.println("Hai " + greeting);
        }
}
```

//applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans-4.2.xsd">
        <bean id="gs1" class="edu.aspire.beans.GreetingServiceImpl" scope="singleton"  lazy-init="default">
                <!--  Setter based IOC-- >
                <property name="greeting">
                        <value>Good Morning</value>
                </property>
        </bean>
        <bean id="gs2" class="edu.aspire.beans.GreetingServiceImpl">
                <!--  Constructor based IOC -- >
                <constructor-arg>
                        <value>Good Evening</value>
                </constructor-arg>
        </bean>
</beans>
```

// HelloApp.java

```
package edu.aspire.test;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.FileSystemResource;
import edu.aspire.beans.GreetingService;
public class HelloClient {
        public static void main(String[] args) {
                // Spring Container
                BeanFactory factory = new XmlBeanFactory(new
                                FileSystemResource("spring_hello/applicationContext.xml"));

                //Spring Framework
                //ApplicationContext factory = new ClassPathXmlApplicationContext("applicationContext.xml");

                GreetingService gs1 = (GreetingService) factory.getBean("gs1");
                GreetingService gs2 = (GreetingService) factory.getBean("gs2");

                gs1.sayGreeting();
                gs2.sayGreeting();
        }
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar
2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar

Additionally following jars are required in case of spring context

4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar

2.ASPECT ORIENTED PROGRAMMING

Aspect Oriented programming (AOP) is often defined as a programming technique that promotes separation of system services (or concerns) from the business logic. The different system services (concerns) are logging, transaction management, exceptions, security, etc. These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system.

The DI helps us decouple our application objects from each other, AOP helps us decouple cross-cutting concerns from the objects that they effect.

AOP makes it possible to modularize these concerns and then apply them declaratively to the components.

The Advantages with AOP

1. The business components look very clean having only business logic without services.
2. All services are implemented in a common place, which simplifies code maintenance.
3. Promotes re-usability.
4. Clear demarcation (separation) among developers.

AOP Terminology

Aspect

Every system service is an aspect, which is written in one place, but can be integrated with components.

Advice

Advice is the actual **implementation of an aspect** i.e., it defines the job that an aspect will perform. Advice defines both **what** and **when** of an aspect.

Spring aspects can work with five kinds of advices:

Advice type	Interface	Description
Before	org.springframework.aop.MethodBeforeAdvice	Called before target method is invoked.
After	org.springframework.aop.AfterAdvice	Called after the target method returns regardless of the outcome.
After-returning	Org.springframework.aop.AfterReturningAdvice	Called after target method completes successfully.
Around	Org.aopalliance.intercept.MethodInterceptor	Both Before and After
After-throwing	org.springframework.aop.ThrowsAdvice	Called when target method throws an exception.

Joinpoint

A joinpoint is a point in the execution of the application where an aspect can be plugged in. Spring only supports method joinpoints.

Pointcut

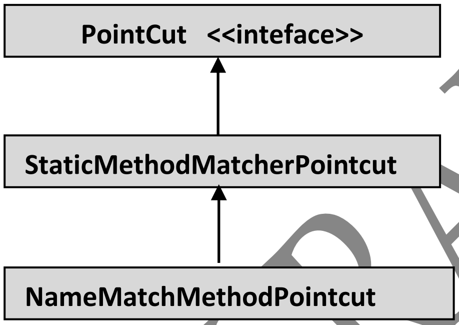
A pointcut defines at what joinpoints advice should be applied, instead of applying advice at all joinpoints i.e., pointcuts help narrow down the join points advised by an aspect. The pointcuts are configured in the spring configuration file.

There are two types of Pointcuts:

I. **Static Pointcut**

Static pointcuts define advice that is always executed.

The public interface org.springframework.aop.PointCut is the root interface for all spring built-in pointcuts. One of built-in static point cut is org.springframework.aop.support.NameMatchMethodPointcut, which is inherited from org.springframework.aop.support.StaticMethodMatcherPointcut.



II. **Dynamic Pointcut**

Dynamic pointcuts determine if advice should be executed by examining the runtime method arguments.

Introduction

An introduction advice allows us to add new methods or attributes to existing classes. For example, we would create an Auditable advice class that keeps the state when an object was last modified. This could be as simple as having one method, setLastModified(Date), and an instance variable to hold this state. This can then be introduced to existing classes without having to change them, giving them new behavior and state.

Target

The target class is a pure business component which is being advised. Without AOP, this class would have to contain its primary logic plus the logic for any cross-cutting concerns. With AOP, the target class is free to focus on its primary logic.

Proxy

A proxy is the object created at runtime after applying advice to the target object.

The proxy <bean> element is defined using org.springframework.aop.framework.**ProxyFactoryBean**.

Proxy = target + advice(s)

Weaving

Weaving is the process of applying aspects to a target object to create a new proxied object.

Advisor

Combining Advice and Pointcut where the advice should be executed is called Advisor.

Advisor = Advice + Pointcut(s)

The public interface PointcutAdvisor the root interface for all built-in Advisors. Most of the spring's built-in pointcuts also have a corresponding PointcutAdvisor.

Built-it Pointcut	Matching Built-in Advisor
NameMatchMethodPointcut	**NameMatchMethodPointcutAdvisor**

Since introduction advice is applied only at class level, introductions have their own advisor: IntroductionAdvisor. Spring also provides default implementation called as DefaultIntroductionAdvisor.

Spring supports following AOPs:

1. Proxy based AOP
2. Declarative based AOP
3. Annotation based AOP

Proxy based AOP

It is a legacy approach hence needs following classes (or interfaces) from Spring API:

1. org.springframework.aop.MethodBeforeAdvice<<interface>>
2. org.springframework.aop.AfterAdvice<<interface>>
3. org.springframework.aop.AfterReturningAdvice<<interface>>
4. org.aopalliance.intercept.MethodInterceptor<<interface>>
5. org.springframework.aop.ThrowsAdvice<<interface>>
6. org.springframework.aop.support.NameMatchMethodPointcut
7. org.springframework.aop.support.NameMatchMethodPointcutAdvisor
8. org.springframework.aop.framework.ProxyFactoryBean

Application #2: Write a Proxy based AOP application using Spring API

```
// Instrument.java
package edu.aspire;
public interface Instrument {
        public void play();
}
//Guitar.java
package edu.aspire;
```

```
public class Guitar implements Instrument {
        public void play() {
                System.out.println("Strum strum strum");
        }
}
//Performer.java
package edu.aspire;
public interface Performer {
        public void perform() throws PerformanceException;
}
//Instrumentlist.java
package edu.aspire;
public class Instrumentalist implements Performer {
       private Instrument instrument;
       public void setInstrument(Instrument instrument) { this.instrument = instrument; }
       public Instrument getInstrument() { return instrument; }
       public void perform() throws PerformanceException { instrument.play(); }
}
// PerformanceException.java
package edu.aspire;
public class PerformanceException extends Exception {
       public PerformanceException() { super(); }
       public PerformanceException(String message) { super(message); }
}

//Audience.java
package edu.aspire;
import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.ThrowsAdvice;
public class Audience implements MethodBeforeAdvice, AfterReturningAdvice, ThrowsAdvice, MethodInterceptor{
       @Override
       public void before(Method method, Object[] data, Object target) throws Throwable {
              System.out.println("The audience is taking their seats.");
       }

      /*@Override
       public void before(Method method, Object[] data, Object target) throws Throwable {
```

```
			System.out.println("The audience is turning off their cellphones");
		}*/

		@Override
		public void afterReturning(Object returnValue, Method method, Object[] data, Object target) throws Throwable{
			System.out.println("CLAP CLAP CLAP CLAP CLAP");
		}
		public void afterThrowing(Method method, Object[] data, Object target, PerformanceException e) {
			System.out.println("Boo! We want our money back!");
		}

		@Override
		public Object invoke(MethodInvocation joinpoint) throws Throwable {
			long start = System.currentTimeMillis();

			Object obj = joinpoint.proceed(); //automatically calls target method

			long end = System.currentTimeMillis();
			System.out.println("***The performance took***:" + (end - start)  + " milliseconds");
			return obj;
		}
}
```

# applicationContext_proxy_based_aop.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-4.2.xsd">
	<!-- Target class -->
	<bean id="target" class="edu.aspire.Instrumentalist">
		<property name="instrument">
			<!—inner bean -- >
			<bean class="edu.aspire.Guitar"/>
		</property>
	</bean>
	<!—advice(s) -- >
	<bean id="audience" class="edu.aspire.Audience" />

	<!—advisor -- >
	<!--
```

```
    <bean id="advisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
            <property name="advice">
                <ref bean="audience"/>
            </property>
            <!-- static point cut -->
            <property name="mappedNames">
                <array>
                    <value>perform</value>
                </array>
            </property>
    </bean> -- >

    <!—Proxy  class -- >
    <bean id="proxybean" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref bean="target" />
        </property>
        <property name="interceptorNames">
            <array>
                <value>audience</value>
                <!-- <value>advisor </value> -- >
            </array>
        </property>
    </bean>
</beans>
```

```
//Client code
package edu.aspire.test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import edu.aspire.Performer;
public class ClientTest {
    public static void main(String[] args) throws Exception {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "applicationContext_proxy_based_aop.xml");
        Performer pRef = (Performer)context.getBean("proxybean");
        pRef.perform();
    }
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar
2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar
4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar
6) **%SPRING_HOME%\libs\spring-aop-4.2.5.RELEASE.jar**
7) **%HOME%\Softwares\jars\aopalliance.jar**

Dis Advantages

Working with Proxy Based AOP using ProxyFactoryBean is overcomplicated when compared to Declarative or Annotation Based AOP.

Defining Pointcuts

In Spring AOP, pointcuts are defined by using AspectJ's pointcut Expression Language, which is used in both Declarative based AOP and Annotation based AOP. The following pointcut designators from AspectJ supported in Spring AOP:

AspectJ designator	Description
Args()	Limits joinpoint matches to the execution of methods whose arguments are instances of the given types.
@args()	Limits joinpoint matches to the execution of methods whose arguments are annotated with the given annotation types.
execution()	Matches joinpoints that are method executions
This()	Limits joinpoint matches to those where the bean reference of the AOP proxy is of a given type
Target()	Limits joinpoint matches to those where the target object is of given type.
@target()	Limits matching to joinpoints where the class of the executing object has an annotation of the given type.
Within()	Limits matching to joinpoints within certain types
@within()	Limits matching to joipints within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
@annotation	Limits joinpoint matches to those where the subject of the joipoint has the given annotation.

The **execution** designator is the only one that actually performs matches. The other designators are used to limit those matches. Hence, the execution is the primary designator we'll use in every pointcut definition we write. The following pointcut expression is used to apply advice whenever Performer's perform() method is executed:

execution(* edu.aspire.Perfomer.perform(..))

We used the execution() designator to select the Perfomer's perform() method. The asterisk indicates that any return type is acceptable. Then we specify the fully qualified class name and the name of the method we want to select. For the method's parameter list, we use the double dot (..), indicates that the pointcut should select any perform() method irrespective of the argument list.

Let's suppose that we want to confine the reach of that pointcut to only the edu.aspire package. In that case, we could limit the match by taking on a **within()** designator as shown below:

execution * edu.aspire.Perfomer.perform(..) && within(edu.aspire.*)

The bean() designator identifies beans by their ID within a pointcut expression. The bean() designator takes a bean ID or name as an argument and limits the piontcut's effect to that specific bean.

execution (* edu.aspire.Perfomer.perform(..) and bean(eddie))

Declarative based AOP

Working with ProxyFactoryBean is difficult. The following elements are from spring's aop configuration namespace:

AOP configuration element	Purpose
<aop:before>	Defines AOP before advice
<aop:after>	Called after target method returns regardless of outcome.
<aop:after-returning>	Called after target method completes successfully.
<aop:around>	Both before and after.
<aop:after-throwing>	Called when target object method throws an exception.
<aop:pointcut>	Defines a pointcut.
<aop:advisor>	Defines a advisor
<aop:aspect>	Defines aspect
<aop:config>	The top-level AOP element.
<aop:aspectj-autoproxy/>	Enables annotation driven aspects using @AspectJ

We see that Audience contained all of the functionality needed for an audience, but none of the details to make it as aspect. That left us having to declare advice and pointcuts in XML.

Application #3: Declarative based AOP

The following files are identical as before.

1. Instrument.java
2. Guitar.java
3. Performer.java
4. Instrumentlist.java
5. PerformanceException.java

```
//Audience1.java
package edu.aspire;
import org.aspectj.lang.ProceedingJoinPoint;
public class Audience1{
	public void takeSeats() {
		System.out.println("The audience is taking their seats.");
```

```
    }
    public void turnOffCellPhones() {
        System.out.println("The audience is turning off their cellphones");
    }
    public void applaud() {
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }
    public void demandRefund() {
        System.out.println("Boo! We want our money back!");
    }
    public void watchPerformance(ProceedingJoinPoint joinpoint) {
        try {
            long start = System.currentTimeMillis();

            joinpoint.proceed();

            long end = System.currentTimeMillis();
            System.out.println("The performance duration is :" + (end - start)  + " milliseconds");
        } catch (Throwable e) {
            System.out.println("Boo! We want our money back!");
        }
    }
}
```

# applicationContext_declarative_based_aop.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">
    <!-- target -->
    <bean id="eddie" class="edu.aspire.Instrumentalist1">
        <property name="instrument">
            <!-- inner bean -->
            <bean class="edu.aspire.Guitar1" />
        </property>
    </bean>
```

```
        <!—audience class -->
        <bean id="audience" class="edu.aspire.Audience1" />

        <aop:config>
                <aop:aspect ref="audience">
                        <!-- Named pointcut to eliminate redundant pointcut definitions -->
                        <aop:pointcut id="performance"
                                        expression="execution(* edu.aspire.Performer1.perform(..))" />
                        <aop:before pointcut-ref="performance" method="takeSeats" />
                        <aop:before pointcut-ref="performance" method="turnOffCellPhones" />
                        <aop:after-returning pointcut-ref="performance" method="applaud" />
                        <!--  <aop:around pointcut-ref="performance" method="watchPerformance" /> -->
                        <!--  <aop:after-throwing pointcut-ref="performance" method="demandRefund" /> -->
                </aop:aspect>
        </aop:config>
</beans>
```

```
//JUnit Test case
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.Performer1;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(“/applicationContext_declarative_based_aop.xml”)
public class AspectTest1{
        @Autowired
        ApplicationContext context;

        @Test
        public void audienceShouldApplaud() throws Exception {
                Performer1 eddie = (Performer1) context.getBean("eddie");
                eddie.perform();
        }
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar
2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar
4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar
6) %SPRING_HOME%\libs\ spring-aop-4.2.5.RELEASE.jar
7) %HOME%\Softwares\jars\aopalliance.jar
8) **%SPRING_HOME%\libs\spring-test-4.2.5.RELEASE.jar**
9) **%HOME%\Softwares\Jars\junit-4.12.jar**
10) **%HOME%\Softwares\Jars\aspectjweaver-1.6.6.jar**

Note: Junit 4 and above additionally needs hamcrest-core.jar

The **@RunWith(SpringJUnit4ClassRunner.class)** tells JUnit to run using spring's test module.
The **@ContextConfiguration**("…xml") is used to specify which spring configuration file to load. This annotation automatically creates appropriate context based on argument.

Annotation based AOP

A key feature introduced in Aspectj 5 is the ability to use annotations to create aspects. The AspectJ's annotation-oriented model makes it simple to turn any class into an aspect by adding few annotations around. This new feature is commonly referred to as **@AspectJ**.
The following files are identical as before.

1. Instrument.java
2. Guitar.java
3. Performer.java
4. Instrumentlist.java
5. PerformanceException.java

But with @AspectJ annotations, we can rewrite Audience class and turn it into an aspect without the need for any additional classes or bean declarations in spring configuration file. The following shows the modified Audience class, now annotated to be an aspect:

Application #4: Example of Annotation based AOP

```
//Audience2.java
package edu.aspire;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
```

```
@Aspect
public class Audience2{  //aspect class
        @Pointcut("execution(* edu.aspire.Performer2.perform(..))")
        public void performance() {}

        @Before("performance()")
        public void takeSeats() { //before adivce
                System.out.println("The audience is taking their seats.....");
        }

        @Before("performance()")
        public void turnOffCellPhones() { //before advice
                System.out.println("The audience is turning off their cellphones");
        }

        @AfterReturning("performance()")
        public void applaud() { //after-returning advice
                System.out.println("CLAP CLAP CLAP CLAP CLAP");
        }

        @AfterThrowing("performance()")
        public void demandRefund() {//after-throwing advice
                System.out.println("Boo! We want our money back!");
        }

        @Around("performance()")
        public void watchPerformance(ProceedingJoinPoint joinpoint) {//around advice
                try {
                        long start = System.currentTimeMillis();

                        joinpoint.proceed();

                        long end = System.currentTimeMillis();
                        System.out.println("***The performance took:" + (end - start) + " milliseconds***");
                } catch (Throwable e) {
                        System.out.println("***Boo! We want our money back***!");
                }
        }
}
```

Because the Audience class contains everything that’s needed to define its own pointcuts and advice, there’s no need to add pointcut and advice declarations in Spring configuration file.
We have to add **<aop:aspectj-autoproxy/>** to turn @Aspectj-annotated classes into proxy advice.

# **applicationContext_annotation_based_aop.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:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">
        <bean id="eddie" class="edu.aspire.Instrumentalist2">
                <property name="instrument">
                        <bean class="edu.aspire.Guitar2" />
                </property>
        </bean>
        <bean id="audience" class="edu.aspire.Audience2" />
        <aop:aspectj-autoproxy />
</beans>
```

//**JUnit Test code**

```
package edu.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.Performer2;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_annotation_based_aop.xml")
public class AspectTest2 {
        @Autowired
        ApplicationContext context;

        @Test
        public void audienceShouldApplaud() throws Exception {
                Performer2 eddie = (Performer2) context.getBean("eddie");
                eddie.perform();
```

```
            }
}
```

Note: Jar files are same as previous application.

3. DEPENDENCY INJECTION

We can reduce spring configuration file size by using different types of injections or wirings such as:

1. Explicit wiring
2. Autowiring (spring 2.0)
3. Annotation based wiring (Newly added in Spring 2.5)
4. Autodiscovery (Newly added in Spring 2.5)

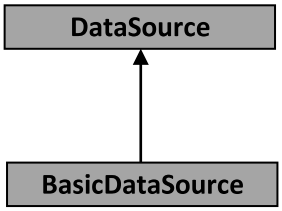
AUTOWIRING

The autowiring eliminate the need for <property> or <constructor-arg> elements by letting spring automatically figure out how to wire bean dependencies.

There are **4** modes of autowirings, they are explained using below bean class:

Example:

```
package edu.aspire.daos;
public class AccountDaoImpl{
        private DataSource dataSource;
        Public AccountDaoImpl(){} //no-arg constructor
        Public AccountDaoImpl(DataSource ds){ this.dataSource = ds;}
        Public void setDataSource(DataSource ds){ this.dataSource = ds;}
        Public DataSource getDataSource(){ return dataSource;}
}
```



Following is the spring configuration file snippet:

```
<beans …>
        <bean id=”dataSource” class=”org.apache.commons.dbcp.BasicDataSource”>
                …
        </bean>
        <bean id=”accdao” class=”edu.aspire.daos.AccountDaoImpl” autowire=”no”>
                <property name=”dataSource”>
                        <ref bean=”dataSource”/>
                </property>
        </bean>
</beans>
```

Below table summarizes four modes of autowiring:

Mode	Description	Example
No	No autowiring means explicit wiring is required using either <property> or <constructor-arg> elements.	<bean id=” accdao” class=” edu.aspire.daos .AccountDaoImpl” **autowire=”no”**> <property name=”dataSource”> <ref bean=”dataSource”/> </property>

		</bean>
Byname	The container automatically wired if the property name in bean class matches with ‘id’ attribute of <bean> element in config file. Unmatched property names will remain unwired.	<bean id=” accdao” class=”AccountDaoImpl” a**utowire=”byName”**/>
byType	Automatically wired if the bean types are assignable to the property data types. Unmatched property types remains unwired. **Note**: The ‘id’ or ‘name’ attribute of injecting <bean> element is not required.	<bean id=” accdao” class=”AccountDaoImpl” a**utowire=”byType”**/>
constructor	Automatically wired if the bean types are assignable to the constructor parameter types.	<bean id=” accdao” class=”AccountDaoImpl” a**utowire=”constructor”**/>

The explicit wiring using <property> and <constructor-arg> elements always have more priority than autowiring.

Default autowiring

If all or majority of the <bean> elements uses same autowire, then we can add ‘**default-autowire**’ attribute to the root <beans> element.
<beans… default-autowire=”byName”>…</beans>
We can still override the default on bean-by-bean basis using the ‘autowire’ attribute of <bean> element.

Mixing auto with explicit wiring

We can mix autowiring as well as explicit wiring. Explicit wiring takes precedence of autowiring.

ANNOTATION BASED WIRING

From Spring 2.5, bean dependencies are wired using annotations. Annotation based autowiring is almost same as ‘autowire’ attribute, but it additionally gives fine-grained autowiring, where we can selectively annotate certain properties for autowiring. **In this case, the ‘autowire’ attribute of <bean> element is not required in spring configuration file**. Also, the setter and getter methods not required in Java class.
By default, annotation based wiring is turned off in the spring container. Hence, to enable, we have to add **<context:annotation-config/>** element from spring’s context namespace:
Example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
                            http://www.springframework.org/schema/context
                            http://www.springframework.org/schema/context/spring-context-4.2.xsd">
        <context:annotation-config/>
        <bean id=”dataSource” class=”org.apache.commons.dbcp.BasicDataSource”>
                …
        </bean>
        <bean id=”accdao” class=”edu.aspire.daos.AccountDaoImpl”/>
</beans>
```

The **<context:annotation-config>** tells spring that we intend to use annotation-based wiring.

The Spring2.5 supports following annotations for autowiring, which should be added to bean class:

```
        @Autowired (Spring’s own)
        @Inject (From JSR-330)
        @Resource (From JSR-250)
```

@Autowired

The @Autowired annotation can be added on top of property or constructor or methods (any method) but preferably added on top of property. **The @Autowired annotation will try to perform ‘byType’ autowiring**.

Example #1: Use @Autowired with property

```
        Public class AccountDaoImpl implements AccountDao{
                @Autowired
                Private DataSource dataSource;
        }
```

Example #2: Use @Autowired with setter method.

```
        Public class AccountDaoImpl implements AccountDao{
                Private DataSource dataSource;
                @Autowired
                Public void setDataSource(DataSource ds){}
        }
```

Example #3: Use @Autowired with constructor.

```
        Public class AccountDaoImpl implements AccountDao{
                Private DataSource dataSource;
                @Autowired
                Public AccountDaoImpl(DataSource ds){}
        }
```

Qualifying Ambiguous Dependencies

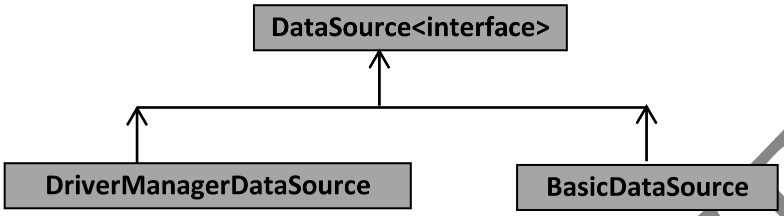
Sometimes more than one bean element may be equally qualified for wiring.

Example:

```
public class AccountDaoImpl{
        @Autowired
        Private DataSource dataSource;
}
```

#applicationContext.xml

```
<beans>
        <context:annotation-config/>
        <bean id=”ds” class=”org.springframework.jdbc.datasource.DriverManagerDataSource”>…</>
        <bean id=”bds” class=”org.apache.commons.dbcp.BasicDataSource”>…</>
        <bean id=”daoimpl” class=”AccountDaoImpl” />
</beans>
```



O/P:

org.springframework.beans.factory.NoSuchBeanDefinitionException: No unique bean of type [javax.sql.DataSource] is defined: expected single matching bean but found 2: [ds, bds]

The **@Qualifier** is used to resolve ambiguous dependencies i.e., it helps @Autowired annotation to choose one of the dependency.

Example:

```
Import org.springframework.bean.factory.annotation.Autowired;
Import org.springframework.bean.factory.annotation.Qualifier;
Public class AccountDaoImpl implements AccountDao{
        @Autowired
        @Qualifier(“bds”)
        private DataSource dataSource;
}
```

The @Qualifier annotation will try to wire in a bean whose ID matches **bds**.

Conclusion: The @Qualifier is really about **narrowing** the selection of autowire candidate beans.

In addition to narrowing by a bean’s ID, it is also possible to narrow by a qualifier that’s is applied to bean itself.

Example:

```
Public class AccountDaoImpl implements AccountDao{
        @Autowired
        @Qualifier(“aspire”)
        Private DataSource dataSource;
}
<bean id=”ds” class=”org.springframework.jdbc.datasource.DriverManagerDataSource”>
<bean id=”bds” class=”org.apache.commons.dbcp.BasicDataSource”>
        <qualifier value=”aspire”/>
        …
```

```
</bean>
```

@Inject

In an effort to unify the programming model among various dependency injection frameworks, sun published dependency injection for java specification (JSR-330). Spring3 supports JSR-330 (also called as Inject model). Instead of using spring-specific @Autowired annotation, we recommended to use @Inject annotation, which is part of JSR-330. The @Inject is a complete replacement for spring specific @Autowired annotation.
Just like @Autowired, the @Inject annotation can be used to autowire properties, methods, and constructors.
We have to download and add **javax.inject.jar** file into the classpath.

```
Import javax.inject.Inject;
Public class AccountDaoImpl implements AccountDao{
        @Inject
        Private DataSource dataSource;
        …
}
```

Add **<context:annotation-config/>** in spring configuration file.
The @Inject's answer to the @Qualifier annotation is the **@Named** annotation to resolve ambiguous dependencies.

Example:

```
Import javax.inject.Inject;
Import javax.inject.Named;
Public class AccountDaoImpl implements AccountDao{
        @Inject
        @Named("bds")
        Private DataSource dataSource;
        …
}
```

Where "**bds**" is bean ID in spring configuration file.

AUTODISCOVERY

Even though <context:annotation-config/> can eliminate <property> and <constructor-arg> elements from spring configuration, we still must explicitly configure beans using<bean> element.
The autodiscovery eliminates <bean> element from spring configuration file.
To configure spring for autodiscovery, use **<context:component-scan/>** instead of <context:annotation-config/>.
The <context:component-scan/> element does everything that <context:annotation-config/> does, plus it automatically discovers beans without adding <bean> element in configuration file.
How does <context:component-scan/> element knows which classes to register as a spring bean?
By default, the <context:component-scan/> looks for classes that are annotated with one of the following stereotype annotations:

1. @Component – Indicates the class is a spring component.

2. @Repository – Indicates class represents data repository. It is used in model layer.
3. @Service – Indicates a service class means business class. It is used in Service layer.
4. @Controller – Indicates Spring MVC controller. It is used in controller layer to handle http requests.

Example:

```
Package edu.aspire.daos;
Import org.springframework.stereotype.Component;
Import org.springframework.bean.factory.annotation.Autowired;
//@Component("accdao")
@Repository("accdao")
Public class AccountDaoImpl implements AccountDao{
        @Autowired
        Private DataSource dataSource;
        …
}
```

Where "accdao" automatically becomes bean ID.
Add **<context:component-scan base-package="edu.aspire.daos"/>** element in the configuration file, which scans package and all of its subpackages.

Note:

1. The '**autowire'** attribute eliminates both <property> as well as <constructor-arg> elements, but still needs <bean> element.
2. The **annotation based wiring** additionally eliminates 'autowire' attribute, but still needs <bean> element.
3. The **autodiscovery** eliminates <bean> element itself.

Java Based Configugration (Spring 3.0)

In Spring 3, Java Based Configuration is included in core Spring module, it allows developer to move bean definition and Spring configuration out of XML file and into Java class.
But, we are still allowed to use the classic XML way to define beans and configuration, the Java Based Configuration is just another alternative solution.
Java based configuration option enables us to write most of our Spring configuration without XML but with the help of few Java-based annotations such as @Configuration, @Bean, etc.
@Configuration: Indicates that the class can be used by the Spring IoC container as a source of bean definitions.
@Bean: Annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

Example #5: Java based configuration

```
package edu.aspire.bean;
public interface GreetingService {
        public void sayGreeting();
}

package edu.aspire.bean;
public class GreetingServiceImpl implements GreetingService {
    private String greeting;

    public GreetingServiceImpl(){  }
    public GreetingServiceImpl(String greeting){   this.greeting = greeting; }

    public void setGreeting(String greeting){  this.greeting = greeting; }

    public void sayGreeting() { System.out.println("Hello "+greeting); }
}

package edu.aspire.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import edu.aspire.beans.GreetingService;
import edu.aspire.beans.GreetingServiceImpl;

@Configuration
public class AppConfig {
    @Bean(name=”gs1”)
    public GreetingService getGs1(){
        GreetingServiceImpl gs = new GreetingServiceImpl();
        gs.setGreeting("Good Morning");
        return gs;
    }

    @Bean(name="gs2")
    public GreetingService getGs2(){
        GreetingServiceImpl gs = new GreetingServiceImpl("Good Afternoon");
        return gs;
    }
}

package edu.aspire.test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import edu.aspire.bean.GreetingService;
import edu.aspire.config.AppConfig;
public class HelloApp {
	public static void main(String[] args) {
		ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

		GreetingService gs1 = (GreetingService) context.getBean("gs1");
		GreetingService gs2 = (GreetingService) context.getBean("gs2");

		gs1.sayGreeting();
		gs2.sayGreeting();
	}
}
```

Note: Jar files are same as previous application.

DIFFERENT TYPES OF CONFIGURATIONS

Spring framework provides flexibility to configure beans in multiple ways such as:

1) XML Based Configuration
2) Java Based Configuration
3) Annotation Based Configuration

Xml Based Configuration	Explicit configuration in XML Ex: applicationContext.xml	<beans>, <bean>, <property>
Java Based Configuration	Explicit configuration in Java Ex: AppConfig.java	@Configuration, @Bean, @Scope
Annotation Based Configuration	Implicit bean discovery and automatic wiring. Directly added to source code such as dao, service, controller classes, etc.	@Component, @Repository, @Service, @Controller

Note: Spring's configuration styles are mix-and-match, so we could choose XML to wire up some beans, use Spring's Java-based configuration (JavaConfig) for other beans, and let remaining beans be automatically discovered by Spring.

Application #6: SPRING-DAO
This application is intended to perform Data Access Object (DAO) operations using pure JDBC API.
This example also includes autowiring.

*/*create table account(ACC_NO NUMBER(4)PRIMARY KEY, ACC_NAME VARCHAR2(20), ACC_TYPE VARCHAR2(20), BAL NUMBER(10,2));*/*

//Account.java

```
package edu.aspire.model;
public class Account{
        private int accno;
        private String accName;
        private String accType;
        private double bal;
        public Account() {}
        public int getAccno() {return accno;}
        public void setAccno(int accno) {this.accno = accno;}
        public String getAccName() {return accName;}
        public void setAccName(String accName) {this.accName = accName;}
        public String getAccType() {return accType;}
        public void setAccType(String accType) {this.accType = accType; }
        public double getBal() { return bal;}
        public void setBal(double bal) {this.bal = bal;}
}
```

//AccountDao.java

```
package edu.aspire.dao;
import edu.aspire.model.Account;
public interface AccountDao {
        public void save(Account account);
        public void update(Account accout);
        public void remove(int accno);
        public Account get(int accno);
}
```

// AccountDaoImpl.java

```
package edu.aspire.dao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.DataSource;
import edu.aspire.model.Account;
import org.springframework.beans.factory.annotation.Required;
```

```
public class AccountDaoImpl implements AccountDao {
    private DataSource dataSource;

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

    @Override
    public void save(Account account) {
        // Declare resources
        Connection con = null;
        PreparedStatement pstmt = null;
        try {
            // Get connection
            con = dataSource.getConnection();
            //Prepare query
            String query = "INSERT INTO account VALUES(?,?,?,?)";
            // Create JDBC statement
            pstmt = con.prepareStatement(query);
            // Set data
            pstmt.setInt(1, account.getAccno());
            pstmt.setString(2, account.getAccName());
            pstmt.setString(3, account.getAccType());
            pstmt.setDouble(4, account.getBal());
            // Execute statement
            pstmt.execute();
        } catch (Exception e) { //Handle Exceptions
            e.printStackTrace();
        } finally {  //Clean up resources to avoid memory leaks problems.
            try {
                pstmt.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public Account get(int accno) {
        // Declare resources
        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        Account avo = null;
        try {
            // Get connection.
            con = dataSource.getConnection();
            // Prepare query.
            String query = "SELECT * FROM account WHERE ACC_NO=" + accno;
            // Create statement object.
            stmt = con.createStatement();
            // Execute query and Process ResultSet
            rs = stmt.executeQuery(query);
            if (rs.next()) {
                avo = new Account();
                avo.setAccno(rs.getInt(1));
                avo.setAccName(rs.getString(2));
                avo.setAccType(rs.getString(3));
                avo.setBal(rs.getDouble(4));
            }
        } catch (Exception e) { //Handle exceptions
            e.printStackTrace();
        } finally { //Release resources to avoid memory leaks.
            try {
                rs.close();
                stmt.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return avo;
    }
    @Override
    public void remove(int accno) {
        // TODO Auto-generated method stub
    }
    @Override
    public void update(Account accout) {
```

```
                // TODO Auto-generated method stub
        }
}
```

# applicationContext_Dao.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context-4.2.xsd">
        <!—Not required from Spring 4.0 -- >
        ~~<context:annotation-config />~~

        <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName">
                        <value>oracle.jdbc.driver.OracleDriver</value>
                </property>
                <property name="url">
                        <value>jdbc:oracle:thin:@localhost:1521:xe</value>
                </property>
                <property name="username">
                        <value>system</value>
                </property>
                <property name="password">
                        <value>manager</value>
                </property>
        </bean>

        <bean id="accdao" class="edu.aspire.dao.AccountDaoImpl" autowire="no">
                <property name="dataSource">
                        <ref bean="dataSource"/>
                </property>
        </bean>

        <!-- <bean id="accdao" class="edu.aspire.dao.AccountDaoImpl" autowire="byName"/> -->
</beans>
```

```
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.model.Account;
import edu.aspire.dao.AccountDao;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_dao.xml")
public class SpringDaoTest {
        @Autowired
        ApplicationContext context;

        @Test
        public void testSpringDao() {
                AccountDao accDao = (AccountDao)context.getBean("accdao");
                Account account = new Account();
                account.setAccno(1);
                account.setAccName("Aspire");
                account.setAccType("Current");
                account.setBal(1000.00);

                accDao.save(account);
        }
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar
2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar
4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar
6) %SPRING_HOME%\libs\ spring-aop-4.2.5.RELEASE.jar
7) %HOME%\Softwares\jars\aopalliance.jar
8) %SPRING_HOME%\libs\spring-test-4.2.5.RELEASE.jar
9) %HOME%\Softwares\Jars\junit-4.9.jar
10) %HOME%\Softwares\Jars\aspectjweaver-1.6.6.jar
11) **%SPRING_HOME%\libs\spring-jdbc-4.2.5.RELEASE.jar**

12) %HOME%\Softwares\Jars\ojdbc6.jar

Application #7: Spring – JDBC
This application is intended to perform database operations using JdbcTemplate. This application also includes "Annotation Based Wiring".

Note: SimpleJdbcTemplate deprecated from Spring 3.1

/* create table employee(eno number(4)primary key, ename varchar2(100), desig varchar2(100), sal number(6,2)); */

//Employee.java

```
package edu.aspire.model;
public class Employee{
        private int eno;
        private String ename;
        private String desig;
        private double sal;
        public Employee() { }
        public int getEno() { return eno; }
        public void setEno(int eno) { this.eno = eno; }
        public String getEname() { return ename; }
        public void setEname(String ename) { this.ename = ename; }
        public String getDesig() { return desig; }
        public void setDesig(String desig) { this.desig = desig; }
        public double getSal() { return sal; }
        public void setSal(double sal) { this.sal = sal; }
}
```

//EmployeeDao.java

```
Package edu.aspire.daos;
import edu.aspire.model.Employee;
public interface EmployeeDao {
        public void save(Employee e);
        public void update(Employee e);
        public void delete(int eno);
        public Employee get(int eno);
}
```

//EmployeeDaoImpl.java

```
package edu.aspire.daos;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import org.springframework.beans.factory.annotation.Autowired;
import edu.aspire.model.Employee;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;

public class EmployeeDaoImpl implements EmployeeDao {
    //Default value is true. In case of 'true', the attribute is injected forcefully to bean in Spring Configuration file
    //In case of false, spring will leave the reference null if no qualifying bean is found.
    @Autowired(required=true)
    private JdbcTemplate jdbcTemplate;

    @Override
    public void save(final Employee emp) {
        //Anonymous approach
        jdbcTemplate.update(new PreparedStatementCreator() {
            public PreparedStatement createPreparedStatement(Connection con)
                        throws SQLException {
                String query = "INSERT INTO employee(eno, ename, desig, sal) VALUES (?,?,?,?)";
                PreparedStatement pstmt =con.prepareStatement(query);
                pstmt.setInt(1, emp.getEno());
                pstmt.setString(2, emp.getEname());
                pstmt.setString(3, emp.getDesig());
                pstmt.setDouble(4, emp.getSal());
                return pstmt;
            }
        });

        //Short cut approach
        /*String query = "INSERT INTO employee(eno, ename, desig, sal) VALUES (?,?,?,?)";
        Object[] data = {emp.getEno(), emp.getEname(), emp.getDesig(), emp.getSal()};
        jdbcTemplate.update(query, data);*/
    }
    @Override
    public Employee get(int eno) {
        String query = "SELECT * FROM EMPLOYEE WHERE ENO=?";
        Employee emp = jdbcTemplate.queryForObject(query, new EmployeeRowMapper(), eno);
        return emp;
    }
    @Override
```

```
        public void delete(int eno) {
                // TODO Auto-generated method stub
        }
        @Override
        public void update(Employee e) {
                // TODO Auto-generated method stub
        }
}
```

# **applicationContext_Jdbc.xml**

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd">
        <bean class="org.apache.commons.dbcp.BasicDataSource" lazy-init=”false”>
                <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
                <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
                <property name="username" value="system" />
                <property name="password" value="manager" />
                <property name="initialSize" value="5" />
                <property name="maxActive" value="10" />
        </bean>

        <bean class="org.springframework.jdbc.core.JdbcTemplate" autowire="constructor"/>

        <bean id="empdao" class="edu.aspire.daos.EmployeeDaoImpl" />
</beans>
```

//SpringJdbcTestjava

```
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.daos.EmployeeDao;
import edu.aspire.model.Employee;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_jdbc.xml")
```

```
public class SpringJdbcTest {
        @Autowired
        ApplicationContext context;
        @Test
        public void testSpringJdbc() {
                EmployeeDao empdao = (EmployeeDao)context.getBean("empdao");

                Employee e = new Employee();
                e.setEno(1);
                e.setEname("Ramesh");
                e.setDesig("Director");
                e.setSal(1000.0);
                empdao.save(e);
                System.out.println("Record inserted successfully...");

                System.out.println("Read customer details");
                Employee emp = empdao.get(1);
               System.out.println(emp.getEno()+"\t"+emp.getEname()+"\t"+emp.getDesig()+"\t"+emp.getSal());
        }
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar
2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar
4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar
6) %SPRING_HOME%\libs\ spring-aop-4.2.5.RELEASE.jar
7) %HOME%\Softwares\jars\aopalliance.jar
8) %SPRING_HOME%\libs\spring-test-4.2.5.RELEASE.jar
9) %HOME%\Softwares\Jars\junit-4.9.jar
10) %HOME%\Softwares\Jars\aspectjweaver-1.6.6.jar
11) %SPRING_HOME%\libs\spring-jdbc-4.2.5.RELEASE.jar
12) %HOME%\Softwares\Jars\ojdbc6.jar
13) %HOME%\Softwares\libs\spring-tx-4.2.5.RELEASE.jar
14) %HOME%\Softwares\Jars\jakarta-commons\commons-dbcp.jar
15) %HOME%\Softwares\Jars\jakarta-commons\commons-pool.jar

Application #8: Spring with Hibernate

This application is intended to integrate Spring with Hibernate Framework.

This application also includes “Autodiscovery” and “Externalizing the configuration”.

```
/* CREATE TABLE CUSTOMER(CNO NUMBER(5)PRIMARY KEY, CNAME VARCHAR2(20), ADDRESS VARCHAR2(100),
PHONE NUMBER(15));*/
package edu.aspire.pojos;
import java.io.Serializable;
public class Customer implements Serializable {
        private int cno;
        private String cname;
        private String address;
        private long phone;
        public Customer() {   }

        public int getCno() { return cno; }
        public void setCno(int cno) { this.cno = cno; }
        public String getCname() { return cname; }
        public void setCname(String cname) { this.cname = cname; }
        public String getAddress() { return address; }
        public void setAddress(String address) { this.address = address; }
        public long getPhone() { return phone; }
        public void setPhone(long phone) { this.phone = phone; }
}
```

#Customer.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="edu.aspire.pojos.Customer" table="CUSTOMER">
                <id name="cno" column="CNO" type="integer" >
                        <generator class="assigned"/>
                </id>
                <property name="cname" column="CNAME" type="string" length="20"/>
                <property name="address" column="ADDRESS" type="string" length="100"/>
                <property name="phone" column="PHONE" type="long"/>
        </class>
</hibernate-mapping>
```

//CustomerDao.java

```
package edu.aspire.daos;
import edu.aspire.pojos.Customer;
public interface CustomerDao {
        public void save(Customer e);
        public void update(Customer e);
        public void delete(int eno);
        public Customer get(int eno);
}
```

//CustomerDaoImpl.java

```
package edu.aspire.daos;
import java.sql.SQLException;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.HibernateTemplate;
//import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;
import edu.aspire.pojos.Customer;

//@Component("custdao")
@Repository(“custdao”)
public class CustomerDaoImpl implements CustomerDao {
        @Autowired
        private HibernateTemplate hibernateTemplate;

        public CustomerDaoImpl() {}
        @Override
        public void save(final Customer c) {
                //anonymous approach
                hibernateTemplate.execute(new HibernateCallback() {
                        @Override
                        public Object doInHibernate(Session session)
                                        throws HibernateException, SQLException {
                                Integer iRef = (Integer) session.save(c);
                                return iRef;
                        }
                });
                // short cut approach
                //hibernateTemplate.save(c);
        }

        @Override
        public void delete(int eno) {
        }

        @Override
        public Customer get(int eno) {
                return null;
        }

        @Override
        public void update(Customer e) {
        }
}
```

#connection.properties

```
jdbc.driverClass=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
jdbc.username=system
jdbc.password=manager
jdbc.initPoolSize=5
jdbc.maxPoolSize=10
```

#applicationContext_Hibernate.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd">
        <context:component-scan base-package="edu.aspire.daos" />
        <!-- Externalizing the configuration -->
        <bean class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
                <property name="locations">
                        <array>
```

```
            <value>connection.properties</value>
        </array>
    </property>
</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClass}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
    <property name="initialSize" value="${ jdbc.initPoolSize }" />
    <property name="maxActive" value="${ jdbc.maxPoolSize }" />
</bean>

<bean class="org.springframework.orm.hibernate3.LocalSessionFactoryBean" >
    <property name="dataSource" ref="dataSource" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.use_sql_comments">true</prop>
            <prop key="hibernate.transaction.factory_class">
                org.hibernate.transaction.JDBCTransactionFactory
            </prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
        </props>
    </property>
    <property name="mappingResources">
        <array>
            <value>Customer.hbm.xml</value>
        </array>
    </property>
    <!-- Including all *.hbm.xml files from directory. -->
    <!-- <property name="mappingDirectoryLocations">
          <array>
             <value>classpath:.</value>
          </array>
     </property> -->
</bean>
```

```
    <bean class="org.springframework.orm.hibernate3.HibernateTemplate" autowire="constructor" />

    <!-- Below <bean> element no longer required -->
    <!-- <bean id="custdao" class="edu.aspire.daos.CustomerDaoImpl">
        <property name="hibernateTemplate" ref="hibernateTemplate"/>
    </bean> -->
</beans>
```

//SpringHibernateTest.java

```
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.daos.CustomerDao;
import edu.aspire.pojos.Customer;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_Hibernate.xml")
public class SpringHibernateTest {
    @Autowired
    ApplicationContext context;

    @Test
    public void testSpringHibernate() {
        CustomerDao custDao = (CustomerDao) context.getBean("custdao");
        Customer c = new Customer();
        c.setCno(1);
        c.setCname("ramesh");
        c.setAddress("Ameerpet");
        c.setPhone(7799108899L);
        custDao.save(c);
        System.out.println("Record inserted successfully...");
    }
}
```

Add following Jars to the eclipse classpath:

1) %SPRING_HOME%\ libs\spring-core-4.2.5.RELEASE.jar

2) %SPRING_HOME%\ libs\spring-beans-4.2.5.RELEASE.jar
3) %HOME%\Softwares\Jars\jakarta-commons\commons-logging.jar
4) %SPRING_HOME%\ libs\spring-context-4.2.5.RELEASE.jar
5) %SPRING_HOME%\ libs\spring-expression-4.2.5.RELEASE.jar
6) %SPRING_HOME%\libs\ spring-aop-4.2.5.RELEASE.jar
7) %HOME%\Softwares\jars\aopalliance.jar
8) %SPRING_HOME%\libs\spring-test-4.2.5.RELEASE.jar
9) %HOME%\Softwares\Jars\junit-4.9.jar
10) %HOME%\Softwares\Jars\aspectjweaver-1.6.6.jar
11) %SPRING_HOME%\libs\spring-jdbc-4.2.5.RELEASE.jar
12) %HOME%\Softwares\Jars\ojdbc6.jar
13) %HOME%\Softwares\libs\ spring-tx-4.2.5.RELEASE.jar
14) %HOME%\Softwares\Jars\jakarta-commons\commons-dbcp.jar
15) %HOME%\Softwares\Jars\jakarta-commons\commons-pool.jar
16) **%SPRING_HOME%\ libs\spring-orm-4.2.5.RELEASE.jar**
17) **%HOME%\Softwares\Jars\hibernate jars*.jar**

4. Profiles and Embedded Databases

One of the most challenging things about developing software is **transitioning** an application from one environment to another. Certain environment-specific choices made for development aren't appropriate or won't work when the application transitions from development to production. Database configuration, encryption algorithms, etc are likely to vary across deployment environments.

Consider database configuration, in a development environment, we're likely to use an **embedded database** preloaded with test data.

Generally EmbeddedDatabaseBuilder created DataSource is perfect for development but not for production. Generally production uses either third-party data sources or JNDI datasources.

Profiles are used to decide appropriate bean for appropriate environment at runtime.

Example:

```
#applicationContext_profiles.xml
<beans xmlns="http://www.springframework.org/schema/beans"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
       http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc-4.2.xsd
       http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-4.2.xsd ">
       <context:component-scan base-package="edu.aspire.profiles.daos" />

       <beans profile="dev">
              <jdbc:embedded-database id="dataSource" type="H2">
                     <jdbc:script location="classpath:schema.sql" />
              </jdbc:embedded-database>
       </beans>

       <beans profile="prod">
              <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
                     <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
                     <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
                     <property name="username" value="system" />
                     <property name="password" value="manager" />
                     <property name="initialSize" value="5" />
                     <property name="maxActive" value="10" />
```

```
        </bean>
    </beans>

    <beans>
        <bean class="org.springframework.orm.hibernate3.LocalSessionFactoryBean" autowire="no">
            <property name="dataSource" ref="dataSource" />
            <property name="hibernateProperties">
                <props>
                    <prop key="hibernate.show_sql">true</prop>
                    <prop key="hibernate.format_sql">true</prop>
                    <prop key="hibernate.use_sql_comments">true</prop>
                    <prop key="hibernate.transaction.factory_class">
                        org.hibernate.transaction.JDBCTransactionFactory
                    </prop>
                </props>
            </property>
            <property name="mappingResources">
                <array>
                    <value>Customer2.hbm.xml</value>
                </array>
            </property>
        </bean>
        <bean class="org.springframework.orm.hibernate3.HibernateTemplate" autowire="constructor" />
    </beans>
</beans>
```

```
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.pojos.Customer2;
import edu.aspire.profiles.daos.CustomerDao;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_profiles.xml")
@ActiveProfiles(profiles = {"dev"})
public class SpringProfilesTest {
        @Autowired
        ApplicationContext context;

        @Test
        public void testSpringHibernate() {
                CustomerDao custDao = (CustomerDao) context.getBean("custdao");
                Customer2 c = new Customer2();
                c.setCno(1);
                c.setCname("ramesh");
                c.setAddress("Ameerpet");
                c.setPhone(7799108899L);
                custDao.save(c);
                System.out.println("Record inserted successfully...");
        }
}
```

The beans which are annotated with @Profile will only be created if the prescribed profile is active but the beans that aren't annotated with @Profile will be always created regardless of which profile is active.
Spring offers the **@ActiveProfiles** annotation to let's specify which profile should be active when a test is run.

We can use **EmbeddedDatabaseBuilder** to construct embedded databases such as **H2, HSQL** and **DERBY**.

```
@Bean
public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2)
                .addScript("classpath:schema.sql")
                .addScript("classpath:test-data.sql")
                .build();
}
```

5.Spring Transaction Management

The Spring framework supports both Programmatic and Declarative transactions.

In case of Programmatic transactions, the business component contains both database operations as well as transactional statements.

In case of Declarative transactions, the business component contains only database operations without transactional statements. The transactional statements are implemented as transaction advice and merged using AOP.

Difference between Programmatic and Declarative transactions:

Programmatic Transactions	Declarative Transactions
Business component contains both database operations as well as transactional statements.	Business component contains only database operations without transactional statements.
Give fine grained control since single method may have multiple transactions.	Convenient to use because of AOP.

Propagation Behavior (or Transaction Attributes)

Method1 begins a transaction and invokes method2. When method2 executes, does it run within the method1 transaction or does it execute in a new transaction? The answer depends on the transaction attribute of method2.

A transaction attribute may have one of the following values:

- Required
- RequiresNew
- Supports
- NotSupported
- Mandatory
- Never
- Nested

Required

If the method1 is running within a transaction and invokes the method2, the method2 executes within the method1 transaction. If the method1 is not associated with a transaction, the container starts a new transaction before running the method2.

RequiresNew

If the method1 is running within a transaction and invokes the method2, the container does following steps:

1. Suspends the method1 transaction
2. Starts a new transaction
3. Delegates the call to the method2
4. Resumes the method1 transaction after the method2 completes.

If the method1 is not associated with a transaction, the container starts a new transaction before running the method2.
You should use the RequiresNew attribute when you want to ensure that the method2 always runs within a new transaction.

Mandatory
If the method1 is running within a transaction and invokes the method2, the method2 executes within the method1 transaction. If the method1 is not associated with a transaction, the container throws the TransactionRequiredException.
Use the Mandatory attribute if the method2 must use the transaction of the method1.

NotSupported
If the method1 is running within a transaction and invokes the method2, the container suspends the method1 transaction before invoking the method2. After the method2 has completed, the container resumes the method1 transaction.
If the method1 is not associated with a transaction, the container does not start a new transaction before running the method2.
Use the NotSupported attribute for methods that don't need transactions.

Supports
If the method1 is running within a transaction and invokes the method2, the method2 executes within the method1 transaction. If the method1 is not associated with a transaction, the container does not start a new transaction before running the method2.

Nested
Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rollback individually from the enclosing transction.

Never
If the method1 is running within a transaction and invokes the method2, the container throws a RemoteException. If the method1 is not associated with a transaction, the container does not start a new transaction for method2.

The above Propagation Behavior is set using txTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW) method.

Isolation Level

The spring framework supports following Isolation levels, these are from TransactionDefinition interface:

1) ISOLATION_READ_UNCOMMITTED
2) ISOLATION_READ_COMMITTED

3) ISOLATION_REPEATABLE_READ
4) ISOLATION_SERIALIZABLE
5) ISOLATION_DEFAULT

The above Isolation levels are set using txTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_DEFAULT) method.

Application #9: Spring – Programmatic Transactions
This application is intended to perform programmatic transactions using spring framework.

```
create  table withdraw(accno number(3)primary key, name varchar2(100), amount number(5));
create  table deposit(accno number(3)primary key, name varchar2(100), amount number(5));
insert into withdraw values(1, 'Ramesh', 1000);
insert into deposit  values(1, 'Balaji', 2000);
commit;
```

```
//Withdraw.java
package edu.aspire.pojos;
import java.io.Serializable;
public class Withdraw implements Serializable {
        private int accno;
        private String name;
        private int amount;
        public Withdraw() { }
        public int getAccno() {  return accno; }
        public void setAccno(int accno) { this.accno = accno; }
        public String getName() { return name; }
        public void setName(String name) { this.name = name; }
        public int getAmount() { return amount; }
        public void setAmount(int amount) { this.amount = amount; }
}
```

```
//Deposit.java
package edu.aspire.pojos;
import java.io.Serializable;
public class Deposit implements Serializable {
        private int accno;
        private String name;
        private int amount;
        public Deposit() {  }
```

```
    public int getAccno() { return accno; }
    public void setAccno(int accno) { this.accno = accno; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAmount() { return amount; }
    public void setAmount(int amount) { this.amount = amount; }
}
```

#**Withdraw.hbm.xml**

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="edu.aspire.pojos.Withdraw" table="WITHDRAW">
        <id name="accno" column="ACCNO" type="integer">
            <generator class="assigned" />
        </id>
        <property name="name" column="NAME" type="string" />
        <property name="amount" column="AMOUNT" type="integer" />
    </class>
</hibernate-mapping>
```

#**Deposit.hbm.xml**

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="edu.aspire.pojos.Deposit" table="DEPOSIT">
        <id name="accno" column="ACCNO" type="integer">
            <generator class="assigned"/>
        </id>
        <property name="name" column="NAME" type="string"/>
        <property name="amount" column="AMOUNT" type="integer"/>
    </class>
</hibernate-mapping>
```

//**WithdrawDaoImpl.java**

```
package edu.aspire.prog.daos;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.stereotype.Repository;
import edu.aspire.pojos.Withdraw;

@Repository
public class WithdrawDaoImpl{
        @Autowired
        private HibernateTemplate hibernateTemplate;
        public Withdraw read(int id) {
                return (Withdraw) hibernateTemplate.get(Withdraw.class, new Integer(id));
        }
        public void update(Withdraw w){
                hibernateTemplate.update(w);
        }
}

//DepositDaoImpl.java
package edu.aspire.prog.daos;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.stereotype.Repository;
import edu.aspire.pojos.Deposit;

@Repository
public class DepositDaoImpl{
        @Autowired
        private HibernateTemplate hibernateTemplate;
        public Deposit read(int id) {
                return (Deposit) hibernateTemplate.get(Deposit.class, new Integer(id));
        }
        public void update(Deposit w) {
                hibernateTemplate.update(w);
        }
}

//ITransferMoney.java
package edu.aspire.tx.programmatic;
public interface ITransferMoney {
        public void transfer(final int fromAccNo, final int toAccNo);
}
```

```
//TransferMoney.java
package edu.aspire.tx.programmatic;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;
import edu.aspire.pojos.Deposit;
import edu.aspire.pojos.Withdraw;
import edu.aspire.prog.daos.DepositDaoImpl;
import edu.aspire.prog.daos.WithdrawDaoImpl;

@Service("transferMoney")
public class TransferMoney implements ITransferMoney {
        @Autowired
        private TransactionTemplate txTemplate;
        @Autowired
        private WithdrawDaoImpl withdradao;
        @Autowired
        private DepositDaoImpl depositdao;

        public void transfer(final int fromAccNo, final int toAccNo) {
                // txTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
                // txTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_DEFAULT);
                txTemplate.execute(new TransactionCallback() {
                        public Object doInTransaction(TransactionStatus arg0) {
                                try {
                                        Withdraw w = withdradao.read(fromAccNo);     //SELECT
                                        w.setAmount(w.getAmount() - 100);
                                        withdradao.update(w);                        //UPDATE

                                        //if(true) throw new Exception();

                                        Deposit d = depositdao.read(toAccNo);        //SELECT
                                        d.setAmount(d.getAmount() + 100);
                                        depositdao.update(d);                        //UPDATE
                                } catch (Exception e) {
                                        e.printStackTrace();
                                        arg0.setRollbackOnly();
                                }
                return null;
            }
        });
    }
}
```

# **applicationContext_Programatic_Tx.xml**

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd">
    <context:component-scan base-package="edu.aspire.tx.programmatic  edu.aspire.prog.daos"/>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
        <property name="username" value="system" />
        <property name="password" value="manager" />
        <property name="initialSize" value="5" />
        <property name="maxActive" value="10" />
    </bean>
    <!-- Mixing autowiring with explicit wiring -->
    <bean class="org.springframework.orm.hibernate3.LocalSessionFactoryBean" autowire="byName">
        <!-- <property name="dataSource" ref="dataSource" /> -->
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.use_sql_comments">true</prop>
                <prop key="hibernate.transaction.factory_class">
                    org.hibernate.transaction.JDBCTransactionFactory
                </prop>
            </props>
        </property>
        <property name="mappingResources">
            <array>
                <value>Withdraw.hbm.xml</value>
                <value>Deposit.hbm.xml</value>
```

```
                    </array>
                </property>
            </bean>

            <bean class="org.springframework.orm.hibernate3.HibernateTransactionManager" autowire="constructor" />

            <bean class="org.springframework.transaction.support.TransactionTemplate" autowire="constructor"/>

            <bean class="org.springframework.orm.hibernate3.HibernateTemplate" autowire="constructor" />
</beans>
```

```
// SpringProgTxsTest.java
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.tx.programmatic.ITransferMoney;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_Programatic_Tx.xml")
public class SpringProgTxsTest {
        @Autowired
        ApplicationContext context;

        @Test
        public void testSpringProgTxs() {
                ITransferMoney transMoney = (ITransferMoney)context.getBean("transferMoney");
                transMoney.transfer(1, 1);
                System.out.println("successfully transferred...");
        }
}
```

Note: Jar files are same as previous application.

Application #10: Spring – Declarative Transactions

Implementing Declarative transactions using Annotation based AOP.

The declarative transactions are implemented using AOP.

The declarative transactions are managed using annotations (From Spring 2.5). Add following element from http://www.springframework.org/schema/tx namespace:

<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"/>

The <tx:annotation-driven/> element tells spring to examine all beans in the context and to look for beans that are annotated with **@Transactional**, either at class-level or at the method level.

For every bean that is annotated with **@Transactional**, the <tx:annotation-driven/> will automatically advise it with transaction advice.

The transaction attributes of the advice will be defined by parameters of the @Transactional annotation.

Please note that the following files are Identical as before:

- ✓ **Withdraw.java**
- ✓ **Deposit.java**
- ✓ **Withdraw.hbm.xml**
- ✓ **Deposit.hbm.xml**
- ✓ **WithdrawDaoImpl.java**
- ✓ **DepositDaoImpl.java**
- ✓ **ITransferMoney.java**

```
// TransferMoney.java
package edu.aspire.tx.declarative;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
import edu.aspire.decl.daos.DepositDaoImpl;
import edu.aspire.decl.daos.WithdrawDaoImpl;
import edu.aspire.decl.pojos.Deposit;
import edu.aspire.decl.pojos.Withdraw;

@Service("transfermoney")
@Transactional(propagation = Propagation.REQUIRED)
public class TransferMoney implements ITransferMoney {
        @Autowired
        private WithdrawDaoImpl withdrawdao;

        @Autowired
        private DepositDaoImpl depositdao;

        public void transfer(final int fromAccNo, final int toAccNo) {
```

```
        Withdraw w = withdrawdao.read(fromAccNo);   // SELECT
        w.setAmount(w.getAmount() - 100);
        withdrawdao.update(w);                       // UPDATE

        Deposit d = depositdao.read(toAccNo);       // SELECT
        d.setAmount(d.getAmount() + 100);           // UPDATE
        depositdao.update(d);
    }
}
```

```
<beans  xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.2.xsd">
    <context:component-scan base-package="edu.aspire.decl.daos  edu.aspire.tx.declarative"/>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
        <property name="username" value="system" />
        <property name="password" value="manager" />
        <property name="initialSize" value="5" />
        <property name="maxActive" value="10" />
    </bean>
    <bean class="org.springframework.orm.hibernate3.LocalSessionFactoryBean" autowire="byName">
        <!-- <property name="dataSource" ref="dataSource"/>  -->
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.use_sql_comments">true</prop>
                <prop key="hibernate.transaction.factory_class">
                    org.hibernate.transaction.JDBCTransactionFactory
                </prop>
            </props>
        </property>
```

```
		<property name="mappingResources">
			<array>
				<value>Withdraw_decl.hbm.xml</value>
				<value>Deposit_decl.hbm.xml</value>
			</array>
		</property>
	</bean>

	<bean id="transactionManager"
	class="org.springframework.orm.hibernate3.HibernateTransactionManager" autowire="constructor"/>

	<!-- Declarative tx using Proxy based AOP-->
	<!-- <bean id="proxyBean"
		class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
		<property name="target">
			<bean class="edu.aspire.tx.declarative.TransferMoney"/>
		</property>
		<property name="transactionManager" ref="transactionManager"/>
		<property name="transactionAttributes">
			<props>
				<prop key="transfer">PROPAGATION_REQUIRED</prop>
			</props>
		</property>
	</bean> -->

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

	<bean class="org.springframework.orm.hibernate3.HibernateTemplate" autowire="constructor" />
</beans>
```

```
// SpringDeclTxsTest.java
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.tx.declarative.ITransferMoney;
```

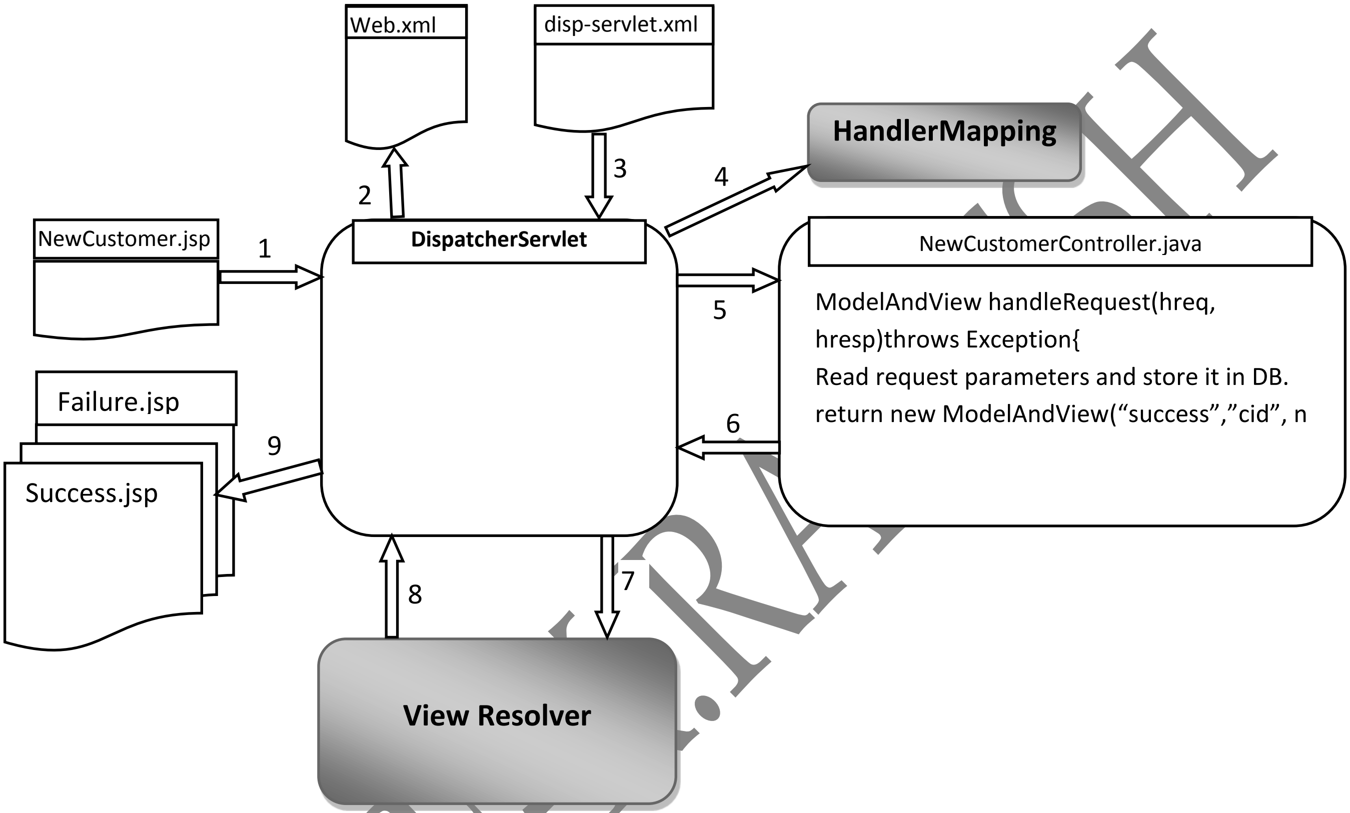
```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext_Declarative_Tx.xml")
public class SpringDeclTxsTest {
	@Autowired
	ApplicationContext context;

	@Test
	public void testSpringDeclTxs() {
		ITransferMoney transMoney = (ITransferMoney) context.getBean("transfermoney");
		transMoney.transfer(1, 1);
		System.out.println("successfully transferred...");
	}
}
```

Note: Jar files are same as previous application.

6. Spring MVC

The Spring MVC helps us to build **web applications** that are **flexible** and **loosely coupled**.


Web.xml
disp-servlet.xml
HandlerMapping
NewCustomer.jsp
DispatcherServlet
NewCustomerController.java
ModelAndView handleRequest(hreq, hresp)throws Exception{
Read request parameters and store it in DB.
return new ModelAndView("success","cid", n
Failure.jsp
Success.jsp
View Resolver
1
2
3
4
5
6
7
8
9

Conclusion: The Spring moves requests around between a dispatcher servlet, handler mappings, controllers, and view resolvers.

Whenever Spring MVC application is deployed into servlet container, then the container will parse web.xml file.

#WEB-INF\web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
        <servlet>
                <servlet-name>disp</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>disp</servlet-name>
                <url-pattern>*.htm</url-pattern>
        </servlet-mapping>
</web-app>
```

In Spring MVC, the spring configuration file name format is: **<servlet-name>-servlet.xml**.
From Servlet 3 specification and in Spring 3.1, we can configure DispatcherServlet in Java file instead of web.xml file.

DispatcherServlet

It is a **front controller** servlet which **delegates** request to controller (business component).
The DispatcherServlet consults HandlerMapping to decide controller class to delegate.
The DispatcherServlet will consult a ViewResolver to map the logical view name to a specific physical response page.

Handler Mappings

It is used to decide one controller among many controllers using urlpath.
The spring api contains following handler mapping implementations:

Handler Mapping Implementation	Description
BeanNameUrlHandlerMapping	Maps controllers to URLs that are based on controllers' bean name in spring configuration file.
ControllerBeanNameHandlerMapping	Same as above except the bean names aren't required to follow URL conventions.
ControllerClassNameHandlerMapping	Maps controllers to URLs by using the controllers' class name as the basis for their URLs.
DefaultAnnotationHandlerMapping	Maps request to controller and controller methods that are annotated with @RequestMapping.
SimpleUrlHandlerMapping	Maps controllers to URLs using a property collection defined in the spring application context.

By default, the DispatcherServlet creates and uses BeanNameUrlHandlerMapping and DefaultAnnotationHandlerMapping. However, we can explicitly configure handler mapping class as below:

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

Controller classes

It is used to provide or invoke **business logic**. Actually, a well-designed controller performs little or no business processing itself and instead delegates responsibility for the business logic to one or more service objects.
It sends back model data along with logical view name to DispatcherServlet.

ViewResolver

The DispatcherServlet consults a view resolver to exchange the logical view name returned by a controller for an actual view that should render the results.
In reality, view resolver job is to map logical view name to some implementation of org.springframework.web.servlet.View.
Spring comes with several view resolver implementations to choose from, as described in below table:

View Resolver	Description

BeanNameViewResolver	Finds an implementation of View that's registered as a <bean> whose ID is same as logical view name. Supports InternalResourceView, VelocityView, FreeMarkerView.
InternalResourceViewResolver	Resolves view name by taking the logical view and surrounding it with a prefix and a suffix. Supports InternalResourceView, and JstlView.
UrlBasedViewResolver	It is a base class for other view resolver classes. Support InternalResourceView, VelocityView, FreeMarkerView.
TilesViewResolver	Looks in tiles definition file (xml file). The definition name and logical view name should be same. Supports TilesView.
ResourceBundleViewResolver	Uses bean defitions in a ResourceBundle (.properties file). Supports InternalResourceView, VelocityView, FreeMarkerView.

The InternalResourceViewResolver takes logical view name and surrounding it with prefix and suffix to resolve response view name including its location.

Example:

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

When controller class returns ‘success’ as a logical view name, then InternalResourceViewResolver prefixes it with ‘/WEB-INF/views’ and suffixes it with ‘.jsp’ and ended up with path as “**/WEB-INF/views/success.jsp**”.

Note: It's good practice to put JSP files that just serve as views under WEB-INF, to hide them from direct access (e.g. via a manually entered URL). Only controllers will be able to access them.

Spring MVC Flow

1) Spring MVC flow starts when user submits request from the browser.
2) The **request** comes to DispatcherServlet via web.xml. It automatically loads the spring configuration file whose name format is <servlet-name>-servlet.xml, where <servlet-name> is configured in web.xml file.
3) The DispatcherServlet **consults** Handler Mapping to decide controller class to delegate.
4) The Handler Mapping **decides** one controller among many controllers based on URL path (/nc.html) specified in request page (NewCustomer.jsp).
5) The DispatcherServlet **delegates** request to the corresponding Controller (CustomerController.java).
6) The controller returns ModelAndView, which contains **logical view** and **model data**, back to DispatcherServlet.
7) The DispatcherServlet **consults** viewresolver to resolve response view name including its location.
8) The viewresolver **returns** physical response view page name.
9) Finally, the DispatcherServlet does view **navigation** to response page (success.jsp).

The Spring MVC flow is almost same as Struts MVC webflow. The minor differences between spring and struts mvc flows are:

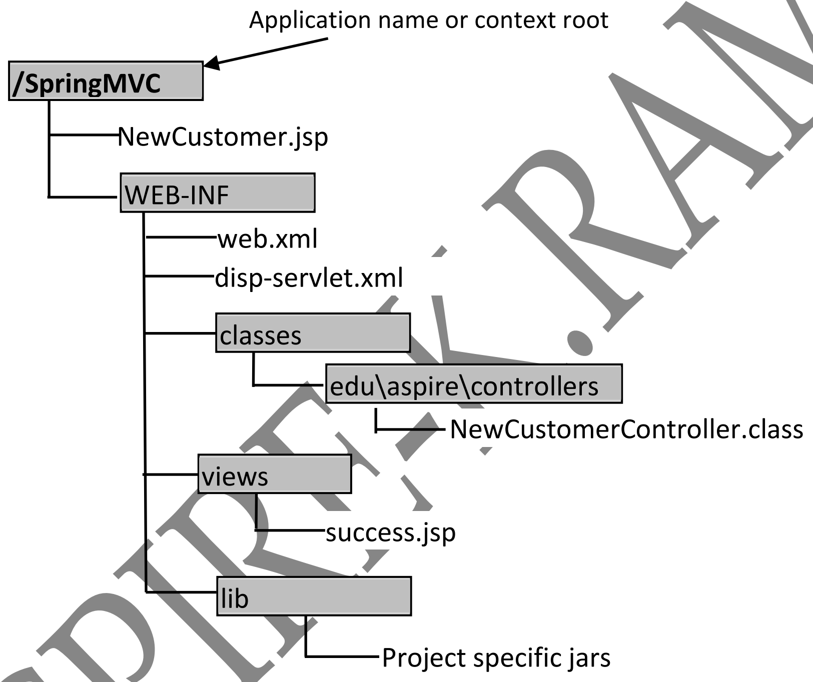
	Spring	Struts
Front-controller	DispatcherServlet	ActionServlet
Business Logic Class	Subclass of Controller Interface.	Subclass of Action class.
Return type	ModelAndView	ActionForward
Method Name	handleRequest()	execute()
FormBeans	No separate formbean	Have separate formbean

Application #11: Spring – MVC

This application is intended to use Spring-MVC module.

Also, the JNDI DataSource is configured to connect with database.

Directory Structure



Note: The following content should be added to **%TOMCAT_HOME%/conf/context.xml**

```
<?xml version='1.0' encoding='utf-8'?>
<Context path="">
<Resource name="mypool" type="javax.sql.DataSource" auth="Container" description=""
maxTotal="15" maxIdle="10" maxWaitMillis="10000" username="system" password="manager"
factory="org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory"
driverClassName="oracle.jdbc.driver.OracleDriver" url="jdbc:oracle:thin:@localhost:1521:xe"/>
</Context>
```

#NewCustomer.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  http://www.w3.org/TR/html4/loose.dtd">
<html><body><pre>
<form action="nc.htm" method="post">
        Name <INPUT type="text" name="cname" />
        Email <INPUT type="text" name="email" />
        Mobile <INPUT type="text" name="mobile" />
        <INPUT type="submit" name="submit" value="Insert" />
</FORM>
</pre></body></html>

//CREATE TABLE CUSTOMER(CID NUMBER(3)PRIMARY KEY, CNAME VARCHAR2(100), EMAIL VARCHAR2(100),
MOBILE VARCHAR2(20));
//CREATE SEQUENCE CUSTOMER_SEQ;
//POJO class
package edu.aspire.domains;
public class Customer {
        private int cid;
        private String cname;
        private String email;
        private long mobile;
        public int getCid() { return cid; }
        public void setCid(int cid) { this.cid = cid; }
        public String getCname() { return cname; }
        public void setCname(String cname) { this.cname = cname; }
        public String getEmail() { return email; }
        public void setEmail(String email) { this.email = email; }
        public long getMobile() { return mobile; }
        public void setMobile(long mobile) { this.mobile = mobile; }
}

#Customer.hbm.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="edu.aspire.domains.Customer" table="CUSTOMER">
                <id name="cid" column="CID" type="integer">
                        <generator class="sequence">
                                <param name="sequence">CUSTOMER_SEQ</param>
```

```
                    </generator>
                </id>
                <property name="cname" column="CNAME" type="string" length="20" />
                <property name="email" column="email" type="string" length="100" />
                <property name="mobile" column="mobile" type="long" />
        </class>
</hibernate-mapping>

//CustomerDao.java
package edu.aspire.model;
import edu.aspire.domains.Customer;
public interface CustomerDao {
        public int save(Customer e);
        public void update(Customer e);
        public void delete(int eno);
        public Customer get(int eno);
}

//CustomerDaoImpl.java
package edu.aspire.model;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.stereotype.Repository;
import edu.aspire.domains.Customer;
@Repository
public class CustomerDaoImpl implements CustomerDao {
        @Autowired(required = true)
        private HibernateTemplate hibernateTemplate;
        public CustomerDaoImpl() { }
        public int save(Customer c) { return (Integer) hibernateTemplate.save(c); }
        public void delete(int eno) { }
        public Customer get(int eno) { return null; }
        public void update(Customer e) { }
}

//NewCustomerController.java
package edu.aspire.controllers;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import edu.aspire.domains.Customer;
import edu.aspire.model.CustomerDao;
public class NewCustomerController implements Controller { //controller class
        @Autowired
        private CustomerDao customerDao;

        @Override
        public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) throws Exception {
                String cname = req.getParameter("cname").trim();
                String email = req.getParameter("email").trim();
                String mobile = req.getParameter("mobile").trim();

                Customer cust = new Customer();
                cust.setCname(cname);
                cust.setEmail(email);
                cust.setMobile(Long.parseLong(mobile));

                Integer cno = (Integer)customerDao.save(cust);
                return new ModelAndView("success", "cid", cno);
        }
}
```

#disp-servlet.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd">
        <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
                <property name="jndiName" value="java:comp/env/mypool" />
        </bean>

        <bean class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
                <property name="dataSource" ref="dataSource" />
                <property name="hibernateProperties">
                <props>
                    <prop key="hibernate.show_sql">true</prop>
                    <prop key="hibernate.format_sql">true</prop>
                    <prop key="hibernate.use_sql_comments">true</prop>
                    <prop key="hibernate.transaction.factory_class">
                        org.hibernate.transaction.JDBCTransactionFactory
                    </prop>
                </props>
            </property>
            <property name="mappingResources">
                <array>
                    <value>Customer.hbm.xml</value>
                </array>
            </property>
    </bean>

    <bean class="org.springframework.orm.hibernate3.HibernateTemplate" autowire="byType" />

    <bean name="/nc.htm" class="edu.aspire.controllers.NewCustomerController" />

    <!-- The below handler mapping is default -->
    <bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

    <!-- Resolving Internal views -->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
            <property name="prefix" value="/WEB-INF/views/" />
            <property name="suffix" value=".jsp" />
            <property name="viewClass" value="org.springframework.web.servlet.view.InternalResourceView" />
            <!-- <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/> -->
    </bean>
</beans>
```

#WEB-INF\views\success.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" http://www.w3.org/TR/html4/loose.dtd">
<html><body>
    Customer ID is: <%=((Integer)request.getAttribute("cid")).intValue()%>
</body> </html>
```

Below java code is alternative for web.xml file:

```
package edu.aspire.config;
```

```
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.XmlWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class AspireWebInitializer implements WebApplicationInitializer{
        @Override
        public void onStartup(ServletContext arg0) throws ServletException {
                /*XmlWebApplicationContext context = new XmlWebApplicationContext();
                  context.setConfigLocation("/WEB-INF/disp-servlet.xml");
                 */
            ServletRegistration.Dynamic dispatcher = arg0.addServlet("disp", new DispatcherServlet());
            dispatcher.setLoadOnStartup(1);
            dispatcher.addMapping("*.htm");

        }
}
```

(**OR**)

```
package edu.aspire.config;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.XmlWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
public class AspireWebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
        @Override
        protected String[] getServletMappings() {
                return new String[] { "*.htm" };
        }

        @Override
        protected Class<?>[] getServletConfigClasses() {
                return new Class<?>[] { AppWebRootConfig.class };
        }
        @Override
        protected Class<?>[] getRootConfigClasses() {
                return null;
        }
}
```

Annotation driven Controllers

This is new feature added in Spring 2.5.
The class is annotated with **@Controller** annotation indicates that this class is controller. Hence, the controller subclass no need to implements Controller interface. The methods annotated with **@RequestMapping** makes method as **request-handling** method. The signature of a **request-handling method can include almost anything as parameter such as HttpServletRequest, HttpServletResponse, Map<String, Object>, Value Object, etc**.
For Example, as a request-handling method, it takes Map<String, Object> as a parameter, which represents the model – the data that's passed between the controller and a response view page. The DispatcherServlet will copy all entries of map into request scope with the same name.
The request-handling method returns logical name of the response view page.
The @Controller annotation is a specialization of the @Component annotation, which means that <context:component-scan/> will pick up and register @Controller annotated classes as beans. To enable autodiscovery, we need to add **<context:component-scan/>** in spring configuration file.
In case of annotation driven controllers, the DispatcherServlet creates and uses **DefaultAnnotationHandlerMapping** from org.springframework.web.servlet.mvc.annotation package. This is good enough for mapping requests to controller methods that are annotated with @RequestMapping.
But we also use other annotations to bind request parameters to handler method parameters, perform validations and perform message conversion. Therefore, the DefaultAnnotationHandlerMapping is not enough. Add **<mvc:annotation-driven/>** element in spring configuration file which registers several features including JSR-303 validation support, message conversion, and support for field formatting.

Form Processing

Working with forms in web applications involve two operations: **displaying the form** and **processing the form submission**. Therefore, in order to register a new Customer in our application, two request-handling methods in CustomerController class are needed to handle each of the operations.

Displaying the registration form

When the form is displayed, it'll need a Customer object to bind to the form fields. The following createCustomerProfile() handler method will create an empty customer object and place it in the model.

```
@Controller
@RequestMapping("/customers")
Public class CustomerController{  //controller class
        @RequestMapping(value="/registration/form")
        public String displayRegistrationForm(Map<String, Object> m){ //endpoint
                m.put("customer", new Customer());
                return "NewCustomer"; //logical name mapped to "/WEB-INF/views/NewCustomer.jsp" page
        }
}
```

The endpoint url to access above method is http://localhost:9090/FormProcessing**/customers/registration/form**

Rendering a form to capture user registration information

The following jsp page uses **Spring's form binding library**
%SPRING_HOME%\projects\org.springframework.web.servlet\src\main\resources\META-INF\spring-forms.tld.

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf"%>
<sf:form action="/FormProcessing/customers/create" method="POST" modelAttribute="customer">
        Full Name: <sf:input path="cname" size="15" />
        E-Mail: <sf:input path="email" size="50"/>
        Mobile: <sf:input path="mobile" size="50"/>
        <input type="submit" value="submit"/>
</sf:form>
```

The <sf:form> tag binds the customer object (identified by the **modelAttribute** attribute) that createCustomerProfile() placed into the model to the various fields in the form. The <sf:input> tags each have a '**path**' attribute that references the property of the Customer object that the form is bound to. When the form is submitted, whatever values these fields contain will be placed into a Customer object and submitted to the server for processing.

Processing form Input

```
@Controller
@RequestMapping("/customers")
Public class CustomerController{
        @RequestMapping(value="/registration/form")
        public String displayRegistrationForm(Map<String, Object> m){  //endpoint
        }

        @RequestMapping(value="/create")
        public String insertCustomer(Customer customer) throws Exception{  // endpoint
                //Read I/P data from vo object
        }
}
```

There are two endpoints to access above methods:

Validations

The **@Valid** annotation is used to trace faulty form input. The @Valid is actually a part of the JavaBean validation specification (JSR-303). The **Spring 3** includes support for JSR-303, and we're using @Valid here to tell Spring that the Customer object should be validated as it is bound to the form input.

The following jars need to be added to classpath:

1) validation-api-1.0.0.GA.jar
2) hibernate-validator-4.3.0.Final.jar
3) jboss-logging-3.0.0.GA.jar

The JSR-303 defines **annotations** that can be **placed on properties** to specify validation rules. The following shows the properties of the Customer class that are annotated with validation annotations.

```
package edu.aspire.view.vo;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
public class Customer {
        private int cid;

        @Size(min = 3, max = 20, message = "Customer Name must be between 3 and 20 characters long.")
        @Pattern(regexp = "^[a-zA-Z  ]+$", message = "Customer Name must be alphabetic")
        private String cname;

        @Pattern(regexp = "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}$", message = "Invalid Email Format")
        private String email;

        @Pattern(regexp="^\\d{10}$", message="Mobile number must be 10-digits numeric number")
        private String mobile;
        …
}
```

Note:

1) ^ shows beginning of string.
2) $ is used for ending the string.
3) \d{10} shows any digit 0-9 having 10 places.

The method signature of handler method is:

```
import javax.validation.Valid;
import org.springframework.validation.Errors;
@RequestMapping(value="/create")
public String insertCustomer(@Valid Customer customer, Errors errors) throws Exception{
        if(errors.hasErrors()){ //In case of validation errors
                return "NewCustomer"; //redirect back to request page
        }
        //Business logic
}
```

If the **hasErrors()** method in Errors class returns true, that means that validation failed. In that case, the method will return logical name of request page (NewCustomer.jsp) as the view name to display the form again so that the user can re-enter and re-submit the form.

Displaying Validation Errors

Use **<sf:errors/>** element from Spring's form binding JSP tag library to render (display) field validation errors.

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf"%>
<sf:form action=”/FormProcessing/customers/create” method="POST" modelAttribute="customer">
        Full Name: <sf:input path="cname" size="15" /><font color=”red”><sf:errors path="cname"/></font>
        E-Mail: <sf:input path="email" size="50"/><font color=”red”><sf:errors path="email"/></font>
        Mobile: <sf:input path="mobile" size="50"/><font color=”red”><sf:errors path="mobile”/></font>
        <input type=”submit” value="submit"/>
</sf:form>
```

The <sf:errors> tag's '**path'** attribute specifies the form field for which errors should be displayed. If there are multiple errors for a single field, they'll all be displayed, separated by an HTML <br/> tag. If you'd rather have them separated some other way, then you can use the delimiter attribute.
We can display all of the errors in one place by using single <sf:errors **path="*"**/>

Application #12: This Case Study includes:

- **a) Annotation driven Controller (@Controller)**
- **b) Spring's form binding library**
- **c) Form Processing**
- **d) Spring's MVC namespace**
- **e) Validations (JSR-303)**
- **f) N-Tier Architecture**
- **g) Http Endpoints**
- **h) Multiple Spring Configuration files**

#WEB-INF\views\NewCustomer.jsp

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf"%>
<html> <body>
<sf:form action=”/FormProcessing/customers/create” method="POST" modelAttribute="customer">
        <fieldset><table cellspacing="0">
                <tr>
                        <th><label for="user_full_name">Full Name:</label></th>
                        <td><sf:input path="cname" size="15" id="user_full_name" />
                            <font color="red"><sf:errors path="cname" delimiter=", "/></font>
```

```
                    </td>
            </tr> <tr>
                    <th><label for="user_email">E-Mail:</label></th>
                    <td><sf:input path="email" size="50" id="user_email" />
                        <font color="red"><sf:errors path="email"/></font>
               </td></tr>
            <tr>
                    <th><label for="user_mobile">Mobile:</label></th>
                    <td><sf:input path="mobile" size="50" id="user_mobile" />
                        <font color="red"><sf:errors path="mobile"/></font>
                    </td>
            </tr><tr>
                    <th></th>
                    <td><input type="submit" value="submit"/></td>
            </tr>
        </table></fieldset>
</sf:form>
</body></html>
```

#Customer.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="edu.aspire.view.vo.Customer" table="CUSTOMER">
                <id name="cid" column="CID" type="integer">
                        <generator class="sequence">
                                <param name="sequence">CUSTOMER_SEQ</param>
                        </generator>
                </id>
                <property name="cname" column="CNAME" type="string" length="20" />
                <property name="email" column="email" type="string" length="100" />
                <property name="mobile" column="mobile" type="string" />
        </class>
</hibernate-mapping>
```

// **CustomerDao.java**

```
package edu.aspire.model;
import edu.aspire.view.vo.Customer;
public interface CustomerDao {
    public int save(Customer e);
    public void update(Customer e);
    public void delete(int eno);
    public Customer get(int eno);
}

// CustomerDaoImpl.java
package edu.aspire.model;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.stereotype.Repository;
import edu.aspire.view.vo.Customer;
//CREATE TABLE CUSTOMER(CID NUMBER(3)PRIMARY KEY, CNAME VARCHAR2(100), EMAIL VARCHAR2(100),
MOBILE VARCHAR2(20));
//CREATE SEQUENCE CUSTOMER_SEQ;
@Repository
public class CustomerDaoImpl implements CustomerDao {
    @Autowired(required = true)
    private HibernateTemplate hibernateTemplate;
    public CustomerDaoImpl() { }
    public int save(Customer c) {  return (Integer) hibernateTemplate.save(c); }
    public void delete(int eno) {  }
    public Customer get(int eno) {  return null; }
    public void update(Customer e) { }
}

// CustomerService.java
package edu.aspire.services;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import edu.aspire.model.CustomerDao;
import edu.aspire.view.vo.Customer;
@Service
public class CustomerService { //business class
    @Autowired
    private CustomerDao customerDao;
    public int processCustomer(Customer c){
        return customerDao.save(c);
    }
}
```

```
//CustomerController.java
package edu.aspire.controllers;
import java.util.Map;
import javax.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import edu.aspire.services.CustomerService;
import edu.aspire.view.vo.Customer;
@Controller
@RequestMapping("/customers")
public class CustomerController{ //Generally controller class handles HTTP requests
        @Autowired
        private CustomerService custService;

        @RequestMapping(value=”/registration/form”, method=RequestMethod.GET)
        public String displayRegistrationForm (Map<String, Object> m){ //endpoint
                m.put("customer", new Customer());
                return "NewCustomer";
        }

        @RequestMapping(value=”/create” method=RequestMethod.POST)
        public String insertCustomer(@Valid Customer customer, Errors errors) throws Exception { //endpoint
                if(errors.hasErrors()){ //In case of validation errors
                        return "NewCustomer"; //redirected back to request page
                }
                Integer cid = custService.processCustomer(customer);
                customer.setCid(cid);
                return "success";
        }
}
```

```
//POJO class / VO class
package edu.aspire.view.vo;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
public class Customer {
	private int cid;
	@Size(min = 3, max = 20, message = "Customer Name must be between 3 and 20 characters long.")
	@Pattern(regexp = "^[a-zA-Z ]+$", message = "Customer Name must be Alphabetic")
	private String cname;
	@Pattern(regexp = "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}$", message = "Invalid Email Format")
	private String email;
	@Pattern(regexp="^\\d{10}$", message="Mobile number must be 10-digits numeric number")
	private String mobile;
	public Customer() { }
	public int getCid() { return cid; }
	public void setCid(int cid) { this.cid = cid; }
	public String getCname() { return cname; }
	public void setCname(String cname) { this.cname = cname; }
	public String getEmail() { return email; }
	public void setEmail(String email) {  this.email = email; }
	public String getMobile() { return mobile; }
	public void setMobile(String mobile) { this.mobile = mobile; }
}
```

#WEB-INF\disp-servlet.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
		xmlns:context="http://www.springframework.org/schema/context"
		xmlns:mvc="http://www.springframework.org/schema/mvc"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
		http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
		http://www.springframework.org/schema/mvc
		http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd
		http://www.springframework.org/schema/context
		http://www.springframework.org/schema/context/spring-context-4.2.xsd" >
	<mvc:annotation-driven />
	<context:component-scan base-package="edu.aspire.controllers" />

	<!-- The below handler mapping is default -->
	<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

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

```
    </bean>
</beans>
```

#WEB-INF/RootConfig.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd ">
    <context:component-scan base-package="edu.aspire.model edu.aspire.services" />
    <!-- Explicit wiring -->
    <bean name="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:comp/env/mypool" />
    </bean>

    <bean class="org.springframework.orm.hibernate3.LocalSessionFactoryBean" autowire="byName">
        <!-- <property name="dataSource" ref="dataSource" /> -->
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.use_sql_comments">true</prop>
                <prop key="hibernate.transaction.factory_class">
                    org.hibernate.transaction.JDBCTransactionFactory
                </prop>
            </props>
        </property>
        <property name="mappingResources">
            <array>
                <value>Customer.hbm.xml</value>
            </array>
        </property>
    </bean>
    <bean class="org.springframework.orm.hibernate3.HibernateTemplate" autowire="byType" />
</beans>
```

#WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
        xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
        <servlet>
                <servlet-name>disp</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>disp</servlet-name>
                <url-pattern>/</url-pattern>
        </servlet-mapping>

        <listener>
                <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
        </listener>
        <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>/WEB-INF/RootConfig.xml </param-value>
        </context-param>
</web-app>
```

#WEB-INF\views\success.jsp

```
<%@page import="edu.aspire.data.Customer"%>
<html><body>
<%
        Customer cust = (Customer) request.getAttribute("customer");
        out.println("Customer ID is:"+cust.getCid());
%>
</body> </html>
```

URL: http://localhost:9090/FormProcessing/customers/registration/form

Any class that extends **AbstractAnnotationConfigDispatcherServletInitializer** will automatically configures DispatcherServlet and ContextLoaderListener i.e., it creates both a DispatcherServlet and a ContextLoaderListener. It will be automatically discovered when deployed in a Servlet 3.0 container.

// **AspireWebAppInitializer.java**

```
package edu.aspire.config;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
public class AspireWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
        @Override
        protected String[] getServletMappings() {
                return new String[] { "/" };
        }
        …
}
```

The **getServletMappings()** identifies one or more paths that DispatcherServlet will be mapped to. The / indicating that it will be the application's default servlet which is DispatherServlet.

The DispatcherServlet is expected to load beans containing web components such as controllers, view resolvers, and handler mappings, but ContextLoaderListener is expected to load the other beans in our application. These beans are typically the middle-tier and data-tier components that drive the back end of the application. Hence in Spring web applications, there's often another application context. This other application context is created by ContextLoaderListener.

The below class is used to load beans containing web components such as controllers, view resolvers, and handler mappings. The DispatcherServlet is expected to load these beans.

```
// AspireWebConfig.java
package edu.aspire.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan("edu.aspire.controllers")
public class AspireWebConfig extends WebMvcConfigurerAdapter {
        @Bean
        public ViewResolver viewResolver() {
                InternalResourceViewResolver resolver = new InternalResourceViewResolver();
                resolver.setPrefix("/WEB-INF/views/");
                resolver.setSuffix(".jsp");
                return resolver;
```

```
    }
}
```

@EanbleWebMvc enables Annotation Based Spring MVC configuration which is same as <mvc:annotation-driven>.

The below configuration class to load the other beans which are typically the middle-tier and data-tier components that drive the back end of the application. The ContextLoaderListener is expected to load these beans.

// **AspireRootConfig.java**

```
package edu.aspire.config;
import java.util.Properties;
import javax.sql.DataSource;
import org.apache.commons.dbcp.BasicDataSource;
import org.hibernate.SessionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.orm.hibernate3.LocalSessionFactoryBean;
import org.springframework.jndi.JndiObjectFactoryBean;
@Configuration
@ComponentScan(basePackages = { "edu.aspire.model", "edu.aspire.services" })
public class AspireRootConfig {
    /*@Bean
    public DataSource dataSource() {
        BasicDataSource bds = new BasicDataSource();
        bds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        bds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        bds.setUsername("system");
        bds.setPassword("manager");
        bds.setInitialSize(10);
        bds.setMaxActive(15);
        return bds;
    }*/
    @Bean
     public DataSource dataSource() {
       JndiObjectFactoryBean dataSource = new JndiObjectFactoryBean();
       dataSource.setJndiName("java:comp/env/mypool");
       try {
         dataSource.afterPropertiesSet(); //Look up the JNDI object and store it.
```

```
      } catch (IllegalArgumentException | NamingException e) {
         throw new RuntimeException(e);
      }
      return (DataSource)dataSource.getObject();
    }

    @Bean
    public LocalSessionFactoryBean sessionFactory(DataSource ds) {
        LocalSessionFactoryBean lsfb = new LocalSessionFactoryBean();
        lsfb.setDataSource(ds);

        Properties props = new Properties();
        props.put("hibernate.show_sql", "true");
        props.put("hibernate.format_sql", "true");
        props.put("hibernate.use_sql_comments", "true");
        props.put("hibernate.transaction.factory_class",  "org.hibernate.transaction.JDBCTransactionFactory");

        lsfb.setHibernateProperties(props);
        lsfb.setMappingResources(new String[] { "Customer.hbm.xml" });
        return lsfb;
    }

    @Bean
    public HibernateTemplate hibernateTemplate(SessionFactory sf) {
        return new HibernateTemplate(sf);
    }
}
```

The @Configuration classes returned from getServletConfigClasses() will define beans for DispatcherServlet's application context.
The @Configuration class's returned from getRootConfigClasses() will be used to configure the application context created by ContextLoaderListener.

```
public class AspireWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { AspireWebConfig.class };
```

```
		}

		@Override
		protected Class<?>[] getRootConfigClasses() {
			return new Class<?>[] { AspireRootConfig.class };
		}
}
```