1. Dependency injection (Inversion of control)

The technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection helps in gluing these classes together and same time keeping them independent.

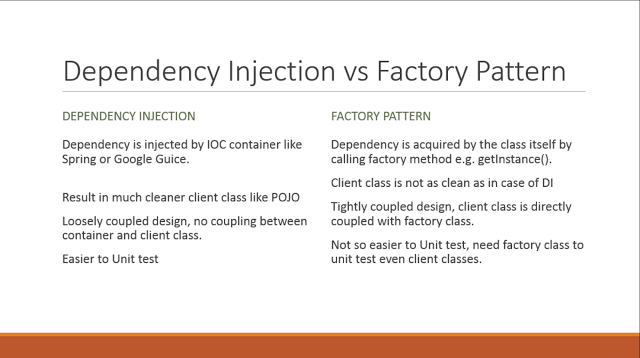
What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent on class B. Now, let's look at the second part, injection. All this means is that class B will get injected into class A by the IoC.

Inversion of Control (IoC) is a general concept, you don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (the IOC container) is then responsible for hooking it all up.

Corner store of Spring DI is Spring Bean Factory.

1. What is difference between Dependency Injection and Factory pattern in Java ?

<http://javarevisited.blogspot.com/2015/06/difference-between-dependency-injection.html>



Factory pattern

In the case of factory design pattern, client class is responsible for calling getInstance() of factory class to create an instance of products, it also means that client class is directly coupled with factory and can't be unit tested without factory class being available.

public class CashRegister {

private PriceCalculator calculator = PriceCalculatorFactory.getInstance();

public void add(Transaction tx) {

int price = calcualtor.getPrice(tx);

add(price);

}

}

In this case dependent class, CashRegister is *directly coupled* with PriceCalculatorFactory because its calling [static getInstance() method](http://java67.blogspot.sg/2012/08/what-is-singleton-pattern-in-java.html) from PriceCalculatorFactory to satisfy its dependency. In order to test CashRegister, you must need a PriceCalculatorFactory, which is not good for unit testing of this class

Dependency Injection

On the other hand in Dependency Injection, client class has no clue about how his dependencies are created and managed. It only knows about dependencies. Mostly dependencies are injected by framework e.g. bean class exists without any hard-coded dependency, as those are injected by IOC container e.g. Spring.

public class CashRegister {

private PriceCalculator calculator;

public CashRegister(PriceCalculator calculator){

this.calculator = calculator;

}

public void add(Transaction tx) {

int price = calcualtor.getPrice(tx);

add(price);

}

public void setCalcuator(PriceCalculator calc){

this.calculator = calc;

}

}

You can see that dependency for CashRegister, which is PriceCalculator is supplied via a constructor, this is known as constructor dependency injection. There is another form of DI as well e.g. setter injection, in which dependency is provided using setter method.

1. What are different types of DI ?

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods.

Also, 3rd way is interface injection

<http://stackoverflow.com/questions/13815139/spring-dependency-injection-for-interfaces>

Interface: 🡪 Step 1: Create an Interface

package org.better.place

public interface SuperDuperInterface{

public void saveWorld();

}

Implementation: 🡪 Step 2: Create implementation class of this interface with @Component

package org.better.place

import org.springframework.stereotype

@Component

public class SuperDuperClass implements SuperDuperInterface{

public void saveWorld(){

System.out.println("Done");

}

}

Client: 🡪 Step 3: Calling implementation class(bean)’s method with @Autowire interface

package org.better.place

import org.springframework.beans.factory.annotation.Autowire;

public class SuperDuperService{

@Autowire

private SuperDuperInterface superDuper;

public void doIt(){

superDuper.saveWorld();

}

}

Now you have your interface defined, written an implementation and marked it as a component - [docs here](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-stereotype-annotations). Now only thing left is to tell spring where to find components so they can be used for autowiring.

<beans ...>

<context:component-scan base-package="org.better.place"/>

</beans>

What is difference between Interface injection and setter method injection ?

<http://programmers.stackexchange.com/questions/163175/difference-between-spring-setter-and-interface-injection>

In interface injection, the setter method is provided by an interface that is implemented by the bean in which we are going to inject the dependency.

1. What is difference between Constructor injection and Setter injection ?

<http://howtodoinjava.com/spring/spring-core/top-spring-core-interview-questions-with-answers/>

1. In Setter Injection, partial injection of dependencies can possible, means if we have 3 dependencies like int, string, long, then its not necessary to inject all values if we use setter injection. If you are not inject it will takes default values for those primitives. In constructor injection, partial injection of dependencies is not possible, because for calling constructor we must pass all the arguments right, if not so we may get error.
2. Setter Injection will overrides the constructor injection value, provided if we write setter and constructor injection for the same property. But, constructor injection cannot overrides the setter injected values. It’s obvious because constructors are called to first to create the instance.
3. In constructor injection, if Object A and B are dependent each other i.e A is depends on B and vice-versa, Spring throws ObjectCurrentlyInCreationException while creating objects of A and B because A object cannot be created until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection because Objects are constructed before setter methods invoked.

Another explainaion of this problem

<http://javarevisited.blogspot.com/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html>

As I said earlier Spring supports both setter and constructor Injection which are two standard way of injecting dependency on beans managed by IOC constructor. Spring framework doesn't support Interface Injection on which dependency is injected by implementing a particular interface. In this section we will see a couple of difference between setter and constructor Injection, which will help you decide when to use setter Injection over constructor Injection in Spring and vice-versa.

1) The fundamental difference between setter and constructor injection, as their name implies is How dependency is injected.  Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container. On the other hand constructor injection uses [constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) to inject dependency on any Spring-managed bean.  
2) Because of using setter method, setter Injection in more readable than constructor injection in Spring configuration file usually applicationContext.xml . Since setter method has name e.g. setReporotService() by reading Spring XML config file you know which dependency you are setting. While in constructor injection, since it uses an index to inject the dependency, it's not as readable as setter injection and you need to refer either Java documentation or code to find which index corresponds to which property.  
3) Another difference between setter vs constructor injection in Spring and one of the drawback of  setter injection is that it does not ensures [dependency Injection](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html). You can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On other hand constructor Injection does not allow you to construct object, until your dependencies are ready.  
4) One more drawback of setter Injection is Security. By using setter injection, you can [override](http://javarevisited.blogspot.in/2011/12/method-overloading-vs-method-overriding.html) certain dependency which is not possible which is not possible with constructor injection because every time you call the constructor, a new object is gets created.  
5) One of our reader Murali Mohan Reddy pointed out one more difference between Setter and Constructor injection in Spring, where later can help if there is a circular dependency between two object A and B.

If Object A and B are dependent each other i.e A is depends ob B and vice-versa. Spring throws ObjectCurrentlyInCreationException while creating objects of A and B bcz A object cannot be created until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection. Objects constructed before setter methods invoked.

When to use constructor and when to use set ?

Setter Injection has upper hand over Constructor Injection in terms of readability. Since for configuring Spring we use [XML files](http://javarevisited.blogspot.in/2011/12/parse-xml-file-in-java-example-tutorial.html), readability is much bigger concern. Also drawback of setter Injection around ensuring mandatory dependency injected or not can be handled by configuring Spring to check dependency using "dependency-check" attribute of  tag or tag. Another worth noting point to remember while comparing Setter Injection vs Constructor Injection is that, once number of dependency crossed a threshold e.g. 5 or 6 its handy manageable to passing dependency via constructor. Setter Injection is preferred choice when number of dependency to be injected is lot more than normal, if some of those arguments is optional than using [Builder design pattern](http://javarevisited.blogspot.in/2012/06/builder-design-pattern-in-java-example.html) is also a good option.

In Summary, both Setter Injection and Constructor Injection has there own advantage and disadvantage. The good thing about Spring is that it doesn't restrict you to use either Setter Injection or Constructor Injection and you are free to use both of them in one Spring configuration file. Use Setter injection when a number of dependencies are more or you need readability. Use Constructor Injection when Object must be created with all of its dependency.

1. What is Spring IoC container ?

Inversion of Control (IoC) is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, the objects define their dependencies that are being injected by other assembler objects.

The Spring IoC creates the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application.

1. What are types of Spring IoC container ?

Bean Factory container: This is the simplest container providing basic support for DI .The BeanFactory is usually preferred where the resources are limited like mobile devices or applet based applications (E.g like XmlBeanFactory is used for reading spring xml configurations)

Spring ApplicationContext Container: This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

(E.g FileSystemXmlApplicationContext: This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.

ClassPathXmlApplicationContext: This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.

WebXmlApplicationContext: This container loads the XML file with definitions of all beans from within a web application.)

1. What is difference between BeanFactory and ApplicationContext ?

<http://howtodoinjava.com/spring/spring-core/top-spring-core-interview-questions-with-answers/>

Major function of BeanFactory

(1)A BeanFactory is like a factory class that contains a collection of beans. The BeanFactory holds Bean Definitions of multiple beans within itself and then instantiates the bean whenever asked for by clients.

(2) BeanFactory is able to create associations between collaborating objects as they are instantiated.

(3) BeanFactory also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods.

Additional function provide by ApplicationContext

1. Text Message resolving and internalization supporting.
2. Generic way to load file.
3. Events to beans register as listener.

Another explaination of difference

<http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html>

|  |  |
| --- | --- |
| **ApplicationContext.** | **BeanFactory** |
| Here we can have more than one config files possible | In this only one config file or .xml file |
| Application contexts can publish events to beans that are registered as listeners | Doesn’t support. |
| Support internationalization (I18N) messages | It’s not |
| Support application life-cycle events, and validation. | Doesn’t support. |
| Supports  many enterprise services such JNDI access, EJB integration, remoting | Doesn’t support. |

Another explaination with XmlBeanFactory

<http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html>

BeanFactory is factory Pattern which is based on IOC design principles.it is used to make a clear separation between application configuration and dependency from actual code. The XmlBeanFactory is one of the implementations of bean Factory which we have used in our project. The org.springframework.beans.factory.xml.XmlBeanFactory is used to create bean instance defined in our XML file.

BeanFactory factory = new XmlBeanFactory(new FileInputStream("beans.xml"));

Or

ClassPathResource resorce = new ClassPathResource("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(resorce);

1. How do you provide configuration metadata to the Spring bean ?

a.XML based configuration file

This is the most popular configuration and we can use bean element in context file to configure a Spring Bean. For example:

<bean name="myBean" class="com.journaldev.spring.beans.MyBean"></bean>

b. Java-based configuration

(1)Based on annotation

If you are using only annotations, you can configure a Spring bean using @Bean annotation. This annotation is used with @Configuration classes to configure a spring bean. Sample configuration is:

@Configuration

@ComponentScan(value="com.journaldev.spring.main")

public class MyConfiguration {

@Bean

public MyService getService(){

return new MyService();

}

}

To get this bean from spring context, we need to use following code snippet:

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(

MyConfiguration.class); 🡪 Note: Use AnnoationConfigApplicaitonContext

MyService service = ctx.getBean(MyService.class);

To enable component scanning, just annotate your @Configuration class as

@ComponentScan(value="com.journaldev.spring.main")

public class MyConfiguration {

...

}

In the example above, the com.journaldev package will be scanned, looking for any @Component annotated classes, and those classes will be registered as Spring bean definitions within the container.

If you are using above configuration in a web application then you will be using AnnotationConfigWebApplicationContext class. This implementation may be used when configuring the Spring ContextLoaderListener servlet listener, Spring MVC DispatcherServlet etc.

(2)Also, if still based on xml, it will be:

@Configuration

public class MyConfiguration {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml"); 🡪 Beans.xml is place to hold definition of beans as same as XML based configuration xml file. 🡪 Note: Use ClassPathXmlApplicationContext

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

}

}

c. Annotation-based configuration

We can also use @Component, @Service, @Repository and @Controller annotations with classes to configure them to be as spring bean. For these, we would need to provide base package location to scan for these classes. For example:

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

1. What is Spring bean ?

Any normal java class that is initialized by Spring IoC container is called Spring Bean. We use Spring ApplicationContext to get the Spring Bean instance.

Spring IoC container manages the life cycle of Spring Bean, bean scopes and injecting any required dependencies in the bean.

1. What are different scopes of Spring bean ?

<http://www.journaldev.com/2696/spring-interview-questions-and-answers>

There are five scopes defined for Spring Beans.

singleton: Only one instance of the bean will be created for each container. This is the default scope for the spring beans. While using this scope, make sure spring bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues because it’s not thread-safe.

prototype: A new instance will be created every time the bean is requested.

request: This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.

session: A new bean will be created for each HTTP session by the container.

global-session: This is used to create global session beans for Portlet applications.

To set spring bean scopes we can use “scope” attribute in bean element or @Scope annotation for annotation based configurations.

1. Does Spring bean provide thread safety ?

The default scope of Spring bean is singleton, so there will be only one instance per context. That means that all the having a class level variable that any thread can update will lead to inconsistent data. Hence in default mode spring beans are not thread-safe.

However we can change spring bean scope to request, prototype or session to achieve thread-safety at the cost of performance.But if your bean has mutable state (e.g. View Model Objects), so you need to ensure thread safety. The most easy and obvious solution for this problem is to change bean scope of mutable beans from “singleton” to “prototype“.

<http://howtodoinjava.com/spring/spring-core/top-spring-core-interview-questions-with-answers/>

1. What is Spring configuration file ?

Spring configuration file is an XML file(servelet-name.xml file). This file contains the classes information and describes how these classes are configured and introduced to each other.

1. Aspect oriented programming (AOP)

One of the key components of Spring is the Aspect oriented programming (AOP) framework. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, and caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Whereas DI helps you decouple your application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect.

The AOP module of Spring Framework provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.

1. AOP concepts

<http://www.journaldev.com/2583/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations>

<http://howtodoinjava.com/spring/spring-aop/top-spring-aop-interview-questions-with-answers/>

1. What are the different advice types in spring?

Advices are actions taken for a particular join point. In terms of programming, they are methods that gets executed when a certain join point with matching pointcut is reached in the application.

Before advice : Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception). To use this advice, use @Before annotation.

After returning advice : Advice to be executed after a join point completes normally. For example, if a method returns without throwing an exception. To use this advice, use @AfterReturning annotation.

After throwing advice : Advice to be executed if a method exits by throwing an exception. To use this advice, use @AfterThrowing annotation.

After advice : Advice to be executed regardless of the means by which a join point exits (normal or exceptional return). To use this advice, use @After annotation.

Around advice : Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. To use this advice, use @Around annotation.

1. What is Spring AOP Proxy?

A proxy is a well-used design pattern. To put it simply, a proxy is an object that looks like another object, but adds special functionality behind the scene.

Spring AOP is proxy-based. AOP proxy is an object created by the AOP framework in order to implement the aspect contracts in runtime.

Spring AOP defaults to using standard JDK dynamic proxies for AOP proxies. This enables any interface (or set of interfaces) to be proxied. Spring AOP can also use CGLIB proxies. This is necessary to proxy classes, rather than interfaces.

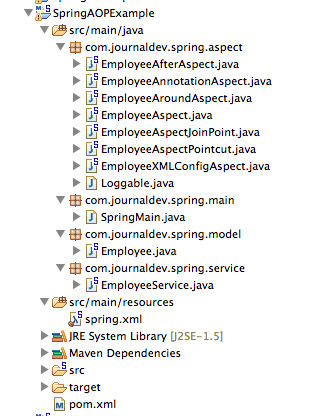
CGLIB is used by default if a business object does not implement an interface.

1. What is Joint point and Point cut?

Join point is a point of execution of the program, such as the execution of a method or the handling of an exception. In Spring AOP, a **join point always represents a method execution**. For example, all the methods defined inside your EmployeeManager interface can be considered joint points if you apply any cross-cutting concern of them.

**Pointcut is a predicate or expression that matches join points.** Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, expression “execution(\* EmployeeManager.getEmployeeById(..))” to match getEmployeeById() the method in EmployeeManager interface). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

1. Example to use AOP



Spring AOP AspectJ Dependencies

Spring framework provides AOP support by default but since we are using AspectJ annotations for configuring aspects and advices, we would need to include them in the pom.xml file.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.springframework.samples</groupId>

<artifactId>SpringAOPExample</artifactId>

<version>0.0.1-SNAPSHOT</version>

<properties>

<!-- Generic properties -->

<java.version>1.6</java.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<!-- Spring -->

<spring-framework.version>4.0.2.RELEASE</spring-framework.version>

<!-- Logging -->

<logback.version>1.0.13</logback.version>

<slf4j.version>1.7.5</slf4j.version>

<!-- Test -->

<junit.version>4.11</junit.version>

<!-- AspectJ -->

<aspectj.version>1.7.4</aspectj.version>

</properties>

<dependencies>

<!-- Spring and Transactions -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring-framework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring-framework.version}</version>

</dependency>

<!-- Logging with SLF4J & LogBack -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4j.version}</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>${logback.version}</version>

<scope>runtime</scope>

</dependency>

<!-- AspectJ dependencies -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>${aspectj.version}</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjtools</artifactId>

<version>${aspectj.version}</version>

</dependency>

</dependencies>

</project>

Model Class

Let’s create a simple java bean that we will use for our example with some additional methods.

Employee.java code:

package com.journaldev.spring.model;

import com.journaldev.spring.aspect.Loggable;

public class Employee {

private String name

public String getName() {

return name;

}

@Loggable

public void setName(String nm) {

this.name=nm;

}

public void throwException(){

throw new RuntimeException("Dummy Exception");

}

}

Did you noticed that setName() method is annotated with Loggable annotation. It is a [custom java annotation](http://www.journaldev.com/721/java-annotations-tutorial-with-custom-annotation-example-and-parsing-using-reflection) defined by us in the project. We will look into it’s usage later on.

Service Class

Let’s create a service class to work with Employee bean.

EmployeeService.java code:

package com.journaldev.spring.service;

import com.journaldev.spring.model.Employee;

public class EmployeeService {

private Employee employee;

public Employee getEmployee(){

return this.employee;

}

public void setEmployee(Employee e){

this.employee=e;

}

}

I could have used Spring annotations to configure it as a Spring Component, but we will use XML based configuration in this project. EmployeeService class is very standard and just provides us an access point for Employee beans.

Spring Bean Configuration with AOP

<?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-4.0.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

<!-- Enable AspectJ style of Spring AOP -->

<aop:aspectj-autoproxy />

<!-- Configure Employee Bean and initialize it -->

<bean name="employee" class="com.journaldev.spring.model.Employee">

<property name="name" value="Dummy Name"></property>

</bean>

<!-- Configure EmployeeService bean -->

<bean name="employeeService" class="com.journaldev.spring.service.EmployeeService">

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

</bean>

<!-- Configure Aspect Beans, without this Aspects advices wont execute -->

<bean name="employeeAspect" class="com.journaldev.spring.aspect.EmployeeAspect" />

<bean name="employeeAspectPointcut" class="com.journaldev.spring.aspect.EmployeeAspectPointcut" />

<bean name="employeeAspectJoinPoint" class="com.journaldev.spring.aspect.EmployeeAspectJoinPoint" />

<bean name="employeeAfterAspect" class="com.journaldev.spring.aspect.EmployeeAfterAspect" />

<bean name="employeeAroundAspect" class="com.journaldev.spring.aspect.EmployeeAroundAspect" />

<bean name="employeeAnnotationAspect" class="com.journaldev.spring.aspect.EmployeeAnnotationAspect" />

</beans>

For using Spring AOP in Spring beans, we need to do following:

1. Declare AOP namespace like xmlns:aop=”http://www.springframework.org/schema/aop”

2. Add aop:aspectj-autoproxy element to enable Spring AspectJ support with auto proxy at runtime

3. Configure Aspect classes as other Spring beans

You can see that I have a lot of aspects defined in the spring bean configuration file, it’s time to look into those one by one.

Spring AOP Before Aspect Example

package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class EmployeeAspect {

@Before("execution(public String getName())")

public void getNameAdvice(){

System.out.println("Executing Advice on getName()");

}

@Before("execution(\* com.journaldev.spring.service.\*.get\*())")

public void getAllAdvice(){

System.out.println("Service method getter called");

}

}

Important points in above aspect class is:

* **Aspect** classes are required to have @Aspect annotation.
* @Before annotation is used to create Before advice
* The string parameter passed in the @Before annotation is the Pointcut expression
* *getNameAdvice()* advice will execute for any Spring Bean method with signature public String getName(). This is a very important point to remember, if we will create Employee bean using new operator the advices will not be applied. Only when we will use ApplicationContext to get the bean, advices will be applied.
* We can use asterisk (\*) as wild card in Pointcut expressions, *getAllAdvice()* will be applied for all the classes in com.journaldev.spring.service package whose name starts with get and doesn’t take any arguments.

Spring AOP Around Aspect Example

As explained earlier, we can use Around aspect to cut the method execution before and after. We can use it to control whether the advised method will execute or not. We can also inspect the returned value and change it. This is the most powerful advice and needs to be applied properly.

EmployeeAroundAspect.java code:

package com.journaldev.spring.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class EmployeeAroundAspect {

@Around("execution(\* com.journaldev.spring.model.Employee.getName())")

public Object employeeAroundAdvice(ProceedingJoinPoint proceedingJoinPoint){

System.out.println("Before invoking getName() method");

Object value = null;

try {

value = proceedingJoinPoint.proceed();

} catch (Throwable e) {

e.printStackTrace();

}

System.out.println("After invoking getName() method. Return value="+value);

return value;

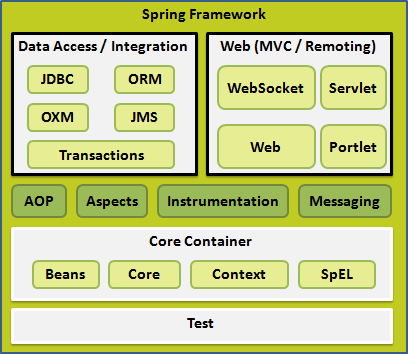
}

}

Around advice are always required to have ProceedingJoinPoint as argument and we should use it’s proceed() method to invoke the target object advised method. If advised method is returning something, it’s advice responsibility to return it to the caller program. For void methods, advice method can return null. Since around advice cut around the advised method, we can control the input and output of the method as well as it’s execution behavior.

1. Modules in Spring

Spring could potentially be a one-stop shop for all your enterprise applications, however, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest.



Core Container:

The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.

The Bean module provides BeanFactory which is a sophisticated implementation of the factory pattern.

The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.

The SpEL module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data access/ Integration:

The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC related coding.

The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.

The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

The JMS module (Java Messaging Service) contains features for producing and consuming messages.

The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web:

The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.

The Web-MVC module contains Spring's model-view-controller (MVC) implementation for web applications.

The Web-Socket module provides support for WebSocket-based, two-way communication between client and server in web applications.

The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous:

The AOP module provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.

The Aspects module provides integration with AspectJ which is again a powerful and mature aspect oriented programming (AOP) framework.

The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers.

The Messaging module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.

The Test module supports the testing of Spring components with JUnit or TestNG frameworks.

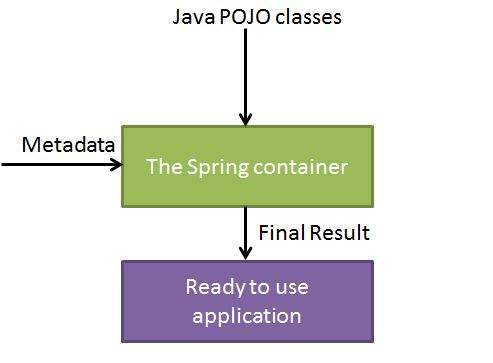
1. IoC container

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application. These objects are called Spring Beans which we will discuss in next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

Spring provides two kinds of container, ApplicationContext and BeanFactory.

The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.



1. Spring bean definition

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container

The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

How to create a bean

Bean's lifecycle details

Bean's dependencies

|  |  |
| --- | --- |
| **Properties** | **Description** |
| Class | This attribute is mandatory and specify the bean class to be used to create the bean. |
| Name | This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). |
| Scope | This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter. |
| constructor-arg | This is used to inject the dependencies and will be discussed in next chapters. |
| Properties | This is used to inject the dependencies and will be discussed in next chapters. |
| autowiring mode | This is used to inject the dependencies and will be discussed in next chapters. |
| lazy-initialization mode | A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup. |
| initialization method | A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter. |
| destruction method | A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter. |

Spring Configuration Metadata

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. There are following three important methods to provide configuration metadata to the Spring Container:

XML based configuration file.

Annotation-based configuration

Java-based configuration

e.g

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- A simple bean definition -->

<bean id="..." class="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with lazy init set on -->

<bean id="..." class="..." lazy-init="true">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with initialization method -->

<bean id="..." class="..." init-method="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with destruction method -->

<bean id="..." class="..." destroy-method="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

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

</beans>

1. Spring bean scopes

When defining a <bean> in Spring, you have the option of declaring a scope for that bean. For example, to force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be prototype. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be singleton.

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.

|  |  |
| --- | --- |
| **Scope** | **Description** |
| singleton | This scopes the bean definition to a single instance per Spring IoC container (default). |
| prototype | This scopes a single bean definition to have any number of object instances. |
| request | This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

The Singleton scope

If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton however, when you need one and only one instance of a bean, you can set the scope property to singleton in the bean configuration file, as shown below:

<!-- A bean definition with singleton scope -->

<bean id="..." class="..." scope="singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

e.g <http://www.tutorialspoint.com/spring/spring_bean_scopes.htm>

Here is the content of HelloWorld.java file:

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.setMessage("I'm object A");

objA.getMessage();

HelloWorld objB = (HelloWorld) context.getBean("helloWorld");

objB.getMessage();

}

}

Following is the configuration file Beans.xml required for singleton scope:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld"

scope="singleton">

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Your Message : I'm object A

Your Message : I'm object A

But as prototype, still use same MainApp and HelloWorld class and the result will be

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld"

scope="prototype">

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Your Message : I'm object A

Your Message : null

1. Bean lifecycle

<http://www.tutorialspoint.com/spring/spring_bean_life_cycle.htm>

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there is lists of the activities that take place behind the scenes between the time of bean Instantiation and its destruction, but this chapter will discuss only two important bean lifecycle callback methods which are required at the time of bean initialization and its destruction.

To define setup and teardown for a bean, we simply declare the <bean> with init-method and/or destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just before a bean is removed from the container.

<http://www.tutorialspoint.com/spring/spring_bean_life_cycle.htm>

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

e.g

Here is the content of HelloWorld.java file:

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

public void init(){

System.out.println("Bean is going through init.");

}

public void destroy(){

System.out.println("Bean will destroy now.");

}

}

Following is the content of the MainApp.java file. Here you need to register a shutdown hook registerShutdownHook() method that is declared on the AbstractApplicationContext class. This will ensures a graceful shutdown and calls the relevant destroy methods.

package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

AbstractApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

context.registerShutdownHook();

}

}

Following is the configuration file Beans.xml required for init and destroy methods:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="helloWorld"

class="com.tutorialspoint.HelloWorld"

init-method="init" destroy-method="destroy">

<property name="message" value="Hello World!"/>

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Bean is going through init.

Your Message : Hello World!

Bean will destroy now.

Which are the important beans lifecycle methods? Can you override them?

<https://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html>

There are two important bean lifecycle methods. The first one is setup which is called when the bean is loaded in to the container. The second method is the teardown method which is called when the bean is unloaded from the container.

The bean tag has two important attributes (init-method and destroy-method) with which you can define your own custom initialization and destroy methods. There are also the correspondive annotations(@PostConstruct and @PreDestroy).

Spring framework provides following 4 ways for controlling life cycle events of bean:

<http://howtodoinjava.com/spring/spring-core/top-spring-core-interview-questions-with-answers/>



1. InitializingBean and DisposableBean callback interfaces

The InitializingBean interface specifies a single method:

void afterPropertiesSet() throws Exception;

This is not a preferrable way to initialize the bean because it tightly couple your bean class with spring container. 🡪 Note: This is not a good way to do

The DisposableBean interface specifies a single method:

void destroy() throws Exception;

A sample bean implementing above interfaces would look like this:

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

public class DemoBeanTypeOne implements InitializingBean, DisposableBean

{

//Other bean attributes and methods

@Override

public void afterPropertiesSet() throws Exception

{

//Bean initialization code

}

@Override

public void destroy() throws Exception

{

//Bean destruction code

}

}

1. Other Aware interfaces for specific behavior
2. custom init() and destroy() methods in bean configuration file

The default init and destroy methods in bean configuration file can be defined in two ways:

1. Bean local definition applicable to a single bean

<beans>

<bean id="demoBean" class="com.howtodoinjava.task.DemoBean" init-method="customInit" destroy-method="customDestroy"></bean>

</beans>

1. Global definition applicable to all beans defined in beans context

<beans default-init-method="customInit" default-destroy-method="customDestroy">

<bean id="demoBean" class="com.howtodoinjava.task.DemoBean"></bean>

</beans>

These methods will be invoked for all bean definitions given under tag. They are useful when you have a pattern of defining common method names such as init() and destroy() for all your beans consistently. This feature helps you in not mentioning the init and destroy method names for all beans independently.

A sample implementation for this type of life cycle will be:

public class BemoBeanTypeThree

{

public void customInit()

{

System.out.println("Method customInit() invoked...");

}

public void customDestroy()

{

System.out.println("Method customDestroy() invoked...");

}

}

1. @PostConstruct and @PreDestroy annotations

@PostConstruct annotated method will be invoked after the bean has been constructed using default constructor and just before it’s instance is returned to requesting object.

@PreDestroy annotated method is called just before the bean is about be destroyed inside bean container.

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

public class BemoBeanTypeFour

{

@PostConstruct

public void customInit()

{

System.out.println("Method customInit() invoked...");

}

@PreDestroy

public void customDestroy()

{

System.out.println("Method customDestroy() invoked...");

}

}

Another explanation of Spring bean life cycle setup

<http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html>

Spring framework is based on IOC so we call it as IOC container also So Spring beans reside inside the IOC container. Spring beans are nothing but Plain old java object (POJO).

Following steps explain their life cycle inside the container.

1. The container will look the bean definition inside configuration file (e.g. bean.xml).
2. Using reflection container will create the object and if any property is defined inside the bean definition then it will also be set.
3. If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean’s ID.
4. If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.
5. If there are any BeanPostProcessors associated with the bean, their post- ProcessBeforeInitialization() methods will be called before the properties for the Bean are set.
6. If an init() method is specified for the bean, it will be called.
7. If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.
8. If the Bean definition in the Configuration file contains a 'destroy-method' attribute, then the corresponding method definition in the Bean class will be called.
9. Bean post processor

<http://www.tutorialspoint.com/spring/spring_bean_post_processors.htm>

The BeanPostProcessor interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the order property provided the BeanPostProcessor implements the Ordered interface.

The BeanPostProcessors operate on bean (or object) instances which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An ApplicationContext automatically detects any beans that are defined with implementation of the BeanPostProcessor interface and registers these beans as post-processors, to be then called appropriately by the container upon bean creation.

Here is the content of HelloWorld.java file:

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

public void init(){

System.out.println("Bean is going through init.");

}

public void destroy(){

System.out.println("Bean will destroy now.");

}

}

This is very basic example of implementing BeanPostProcessor, which prints a bean name before and after initialization of any bean. You can implement more complex logic before and after instantiating a bean because you have access on bean object inside both the post processor methods.

Here is the content of InitHelloWorld.java file:

package com.tutorialspoint;

import org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.beans.BeansException;

public class InitHelloWorld implements BeanPostProcessor {

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

System.out.println("BeforeInitialization : " + beanName);

return bean; // you can return any other object as well

}

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

System.out.println("AfterInitialization : " + beanName);

return bean; // you can return any other object as well

}

}

Following is the content of the MainApp.java file. Here you need to register a shutdown hook registerShutdownHook() method that is declared on the AbstractApplicationContext class. This will ensures a graceful shutdown and calls the relevant destroy methods.

package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

AbstractApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

context.registerShutdownHook();

}

}

Following is the configuration file Beans.xml required for init and destroy methods:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld"

init-method="init" destroy-method="destroy">

<property name="message" value="Hello World!"/>

</bean>

<bean class="com.tutorialspoint.InitHelloWorld" />

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

BeforeInitialization : helloWorld

Bean is going through init.

AfterInitialization : helloWorld

Your Message : Hello World!

Bean will destroy now.

1. Bean definition inheritance

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but inheritance concept is same. You can define a parent bean definition as a template and other child beans can inherit required configuration from the parent bean.

When you use XML-based configuration metadata, you indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute

Following is the configuration file Beans.xml where we defined "helloWorld" bean which has two properties message1 and message2. Next "helloIndia" bean has been defined as a child of "helloWorld" bean by using parent attribute. The child bean inherits message2 property as is, and overrides message1 property and introduces one more property message3.

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld">

<property name="message1" value="Hello World!"/>

<property name="message2" value="Hello Second World!"/>

</bean>

<bean id="helloIndia" class="com.tutorialspoint.HelloIndia" parent="helloWorld">

<property name="message1" value="Hello India!"/>

<property name="message3" value="Namaste India!"/>

</bean>

</beans>

Here is the content of HelloWorld.java file:

package com.tutorialspoint;

public class HelloWorld {

private String message1;

private String message2;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void getMessage1(){

System.out.println("World Message1 : " + message1);

}

public void getMessage2(){

System.out.println("World Message2 : " + message2);

}

}

Here is the content of HelloIndia.java file:

package com.tutorialspoint;

public class HelloIndia {

private String message1;

private String message2;

private String message3;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void setMessage3(String message){

this.message3 = message;

}

public void getMessage1(){

System.out.println("India Message1 : " + message1);

}

public void getMessage2(){

System.out.println("India Message2 : " + message2);

}

public void getMessage3(){

System.out.println("India Message3 : " + message3);

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.getMessage1();

objA.getMessage2();

HelloIndia objB = (HelloIndia) context.getBean("helloIndia");

objB.getMessage1();

objB.getMessage2();

objB.getMessage3();

}

}

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

World Message1 : Hello World!

World Message2 : Hello Second World!

India Message1 : Hello India!

India Message2 : Hello Second World!

India Message3 : Namaste India!

1. Dependency injection

Every java based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor() {

spellChecker = new SpellChecker();

}

}

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario we would instead do something like this:

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

Here TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to TextEditor at the time of TextEditor instantiation and this entire procedure is controlled by the Spring Framework.

Here, we have removed the total control from TextEditor and kept it somewhere else (ie. XML configuration file) and the dependency ( ie. class SpellChecker) is being injected into the class TextEditor through a Class Constructor. Thus flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

Second method of injecting dependency is through Setter Methods of TextEditor class where we will create SpellChecker instance and this instance will be used to call setter methods to initialize TextEditor's properties.

|  |  |
| --- | --- |
| **S.N.** | **Dependency Injection Type & Description** |
| 1 | [**Constructor-based dependency injection**](http://www.tutorialspoint.com/spring/constructor_based_dependency_injection.htm)  Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class. |
| 2 | [**Setter-based dependency injection**](http://www.tutorialspoint.com/spring/setter_based_dependency_injection.htm)  Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean. |

You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

1. Inject inner bean

As you know Java inner classes are defined within the scope of other classes, similarly, inner beans are beans that are defined within the scope of another bean. Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean and it is shown below.

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="outerBean" class="...">

<property name="target">

<bean id="innerBean" class="..."/>

</property>

</bean>

</beans>

Here is the content of TextEditor.java file:

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker

// a setter method to inject the dependency.

public void setSpellChecker(SpellChecker spellChecker) {

System.out.println("Inside setSpellChecker." );

this.spellChecker = spellChecker;

}

// a getter method to return spellChecker

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file SpellChecker.java:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file Beans.xml which has configuration for the setter-based injection but using inner beans:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean using inner bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

<property name="spellChecker">

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker"/>

</property>

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside SpellChecker constructor.

Inside setSpellChecker.

Inside checkSpelling.

1. How can you inject Java Collection into Spring ?

You have seen how to configure primitive data type using value attribute and object references using ref attribute of the <property> tag in your Bean configuration file. Both the cases deal with passing singular value to a bean.

Now what about if you want to pass plural values like Java Collection types List, Set, Map, and Properties.

|  |  |
| --- | --- |
| **Element** | **Description** |
| <list> | This helps in wiring ie injecting a list of values, allowing duplicates. |
| <set> | This helps in wiring a set of values but without any duplicates. |
| <map> | This can be used to inject a collection of name-value pairs where name and value can be of any type. |
| <props> | This can be used to inject a collection of name-value pairs where the name and value are both Strings. |

You can use either <list> or <set> to wire any implementation of java.util.Collection or an array.

You will come across two situations

(a) Passing direct values of the collection and

(b) Passing a reference of a bean as one of the collection elements.

Here is the content of JavaCollection.java file:

package com.tu

torialspoint;

import java.util.\*;

public class JavaCollection {

List addressList;

Set addressSet;

Map addressMap;

Properties addressProp;

// a setter method to set List

public void setAddressList(List addressList) {

this.addressList = addressList;

}

// prints and returns all the elements of the list.

public List getAddressList() {

System.out.println("List Elements :" + addressList);

return addressList;

}

// a setter method to set Set

public void setAddressSet(Set addressSet) {

this.addressSet = addressSet;

}

// prints and returns all the elements of the Set.

public Set getAddressSet() {

System.out.println("Set Elements :" + addressSet);

return addressSet;

}

// a setter method to set Map

public void setAddressMap(Map addressMap) {

this.addressMap = addressMap;

}

// prints and returns all the elements of the Map.

public Map getAddressMap() {

System.out.println("Map Elements :" + addressMap);

return addressMap;

}

// a setter method to set Property

public void setAddressProp(Properties addressProp) {

this.addressProp = addressProp;

}

// prints and returns all the elements of the Property.

public Properties getAddressProp() {

System.out.println("Property Elements :" + addressProp);

return addressProp;

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

JavaCollection jc=(JavaCollection)context.getBean("javaCollection");

jc.getAddressList();

jc.getAddressSet();

jc.getAddressMap();

jc.getAddressProp();

}

}

Following is the configuration file Beans.xml which has configuration for all the type of collections:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for javaCollection -->

<bean id="javaCollection" class="com.tutorialspoint.JavaCollection">

<!-- results in a setAddressList(java.util.List) call -->

<property name="addressList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</list>

</property>

<!-- results in a setAddressSet(java.util.Set) call -->

<property name="addressSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</set>

</property>

<!-- results in a setAddressMap(java.util.Map) call -->

<property name="addressMap">

<map>

<entry key="1" value="INDIA"/>

<entry key="2" value="Pakistan"/>

<entry key="3" value="USA"/>

<entry key="4" value="USA"/>

</map>

</property>

<!-- results in a setAddressProp(java.util.Properties) call -->

<property name="addressProp">

<props>

<prop key="one">INDIA</prop>

<prop key="two">Pakistan</prop>

<prop key="three">USA</prop>

<prop key="four">USA</prop>

</props>

</property>

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

List Elements :[INDIA, Pakistan, USA, USA]

Set Elements :[INDIA, Pakistan, USA]

ap Elements :{1=INDIA, 2=Pakistan, 3=USA, 4=USA}

Property Elements :{two=Pakistan, one=INDIA, three=USA, four=USA}

Injecting Bean References:

Following Bean definition will help you understand how to inject bean references as one of the collection's element. Even you can mix references and values all together as shown below:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Bean Definition to handle references and values -->

<bean id="..." class="...">

<!-- Passing bean reference for java.util.List -->

<property name="addressList">

<list>

<ref bean="address1"/>

<ref bean="address2"/>

<value>Pakistan</value>

</list>

</property>

<!-- Passing bean reference for java.util.Set -->

<property name="addressSet">

<set>

<ref bean="address1"/>

<ref bean="address2"/>

<value>Pakistan</value>

</set>

</property>

<!-- Passing bean reference for java.util.Map -->

<property name="addressMap">

<map>

<entry key="one" value="INDIA"/>

<entry key ="two" value-ref="address1"/>

<entry key ="three" value-ref="address2"/>

</map>

</property>

</bean>

</beans>

To use above bean definition, you need to define your setter methods in such a way that they should be able to handle references as well.

Injecting null and empty string values

If you need to pass an empty string as a value then you can pass it as follows:

<bean id="..." class="exampleBean">

<property name="email" value=""/>

</bean>

The preceding example is equivalent to the Java code: exampleBean.setEmail("")

If you need to pass an NULL value then you can pass it as follows:

<bean id="..." class="exampleBean">

<property name="email"><null/></property>

</bean>

The preceding example is equivalent to the Java code: exampleBean.setEmail(null)

1. What is Beans autowiring and different mode of autowiring ?

The Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements which helps cut down on the amount of XML configuration you write for a big Spring based application.

Autowire modes

|  |  |
| --- | --- |
| **Mode** | **Description** |
| No | This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter. |
| [**byName**](http://www.tutorialspoint.com/spring/spring_autowiring_byname.htm) | Autowiring by property name. Spring container looks at the properties of the beans on which *autowire* attribute is set to*byName* in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. |
| [**byType**](http://www.tutorialspoint.com/spring/spring_autowiring_bytype.htm) | Autowiring by property datatype. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byType* in the XML configuration file. It then tries to match and wire a property if its **type** matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown. |
| [**constructor**](http://www.tutorialspoint.com/spring/spring_autowiring_byconstructor.htm) | Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |
| Autodetect | Spring first tries to wire using autowire by *constructor*, if it does not work, Spring tries to autowire by *byType*. |

1. What are limitations with autowiring ?

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.

|  |  |
| --- | --- |
| **Limitations** | **Description** |
| Overriding possibility | You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring. |
| Primitive data types | You cannot autowire so-called simple properties such as primitives, Strings, and Classes. |
| Confusing nature | Autowiring is less exact than explicit wiring, so if possible prefer using explict wiring. |

1. Spring autowiring by name

This mode specifies autowiring by property name. Spring container looks at the beans on which auto-wire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

For example, if a bean definition is set to autowire byName in configuration file, and it contains a spellChecker property (that is, it has a setSpellChecker(...) method), Spring looks for a bean definition named spellChecker, and uses it to set the property. Still you can wire remaining properties using <property> tags. Following example will illustrate the concept.

Here is the content of TextEditor.java file:

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

private String name;

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file SpellChecker.java:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker() {

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file Beans.xml in normal condition:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

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

<property name="name" value="Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

But if you are going to use autowiring 'byName', then your XML configuration file will become as follows:

ApplicationContext.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor"

autowire="byName">

<property name="name" value="Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside SpellChecker constructor.

Inside checkSpelling.

Autowiring spring bean by name using annotation

<http://stackoverflow.com/questions/11831261/autowiring-spring-bean-by-name-using-annotation>

In Springs latest version, we can autowire a bean using annotation as **@Autowire**. This will autowire the bean using its type(or constructor, if applied on it).  
Is there any way I can use autowire annotation based on the bean name which we were doing without annotaion in spring's xml file as **autowire="byName"**?

You can use @Autowire @Qualifier("beanname")

You can use JSR-250 @Resource for by-name bean autowiring, unless you need constructor injection or multi-parameter method injection.

From the docs:

If you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if is technically capable of referring to a bean name through @Qualifier values. Instead, use the JSR-250 @Resource annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process.

1. Spring autowire bytype

This mode specifies autowiring by property type. Spring container looks at the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

For example, if a bean definition is set to autowire byType in configuration file, and it contains a spellChecker property of SpellChecker type, Spring looks for a bean definition named SpellChecker, and uses it to set the property. Still you can wire remaining properties using <property> tags. Following example will illustrate the concept where you will find no difference with above example except XML configuration file has been changed.

Just change the above bean configuration file as follows:

Applicationcontext.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor"

autowire="byType">

<property name="name" value="Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id="SpellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

1. Spring autowiring by constructor

This mode is very similar to byType, but it applies to constructor arguments. Spring container looks at the beans on which autowire attribute is set to constructor in the XML configuration file. It then tries to match and wire its constructor's argument with exactly one of the beans name in configuration file. If matches are found, it will inject those beans otherwise, it will throw exceptions.

For example, if a bean definition is set to autowire by constructor in configuration file, and it has a constructor with one of the arguments of SpellChecker type, Spring looks for a bean definition named SpellChecker, and uses it to set the constructor's argument. Still you can wire remaining arguments using <constructor-arg> tags. Following example will illustrate the concept.

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor"

autowire="constructor">

<constructor-arg value="Generic Text Editor"/> 🡪 Use constructor-arg

</bean>

<!-- Definition for spellChecker bean -->

<bean id="SpellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

The difference between byType and byName autowiring is as follows :

<http://findnerd.com/list/view/Difference-between-autowire-byName-and-byType-in-Spring/4466/>

Spring framework provides the way to inject the bean dependencies using the autowire functionality. By default autowire functionality is disabled in spring framework. A developer explicitly define the autowiring configuration in spring configuration xml file. And this point is to remember that autowiring is only supported for object dependencies.

1. Autowire byType will search for a bean in configuration file, whose id match with the property type to be wired whereas autowire byName will search for a bean whose id is matching with the property name to be wired.
2. As a syntax wise difference is as follows:

package com.evon;

* 1. **public** **class** Company {
  2. **private** Employee emp;
  3. **public** String setEmp(Employee emp) {
  4. **this** .emp = emp;
  5. }
  6. }

**in configuration file :**

1. <bean id="beanId1" **class**="com.evon.Company " autowire="byType"/>
2. <bean id="employee" **class**="com.evon.Employee ">
3. <property name="empName" value="Rahul" />
4. </bean>

In the above example container will search for a employee bean in the configuration xml file which should be of type Employee.

And following is the example of autowire byName where container will search for a bean which should be register with the id name same as filed name ('emp') in configuration file. Like

1. <bean id="beanId2" **class**="com.evon.Company" autowire="byName"/>
2. <bean id="emp" **class**="com.evon.Employee ">
3. <property name="empName" value="Rahul" />
4. </bean>
5. How to turn on Spring annotation based configuration ?

Starting from Spring 2.5 it became possible to configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So consider to have following configuration file in case you want to use any annotation in your Spring application.

ApplicationContext.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/> 🡪 Note: To use @Autowired annotation in bean classes, you must first enable the annotation

<!-- bean definitions go here -->

</beans>

Once <context:annotation-config/> is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors.

1. Annotation explanation

@Required

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. otherwise the container throws a BeanInitializationException exception. Below is an example to show the use of @Required annotation.

Here is the content of Student.java file:

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Required;

public class Student {

private Integer age;

private String name;

@Required

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Required

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

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

System.out.println("Name : " + student.getName() );

System.out.println("Age : " + student.getAge() );

}

}

Following is the content of the configuration file Beans.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for student bean -->

<bean id="student" class="com.tutorialspoint.Student">

<property name="name" value="Zara" />

<!-- try without passing age and check the result -->

<!-- property name="age" value="11"-->

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will raise BeanInitializationException exception and print the following error along with other log messages: Property 'age' is required for bean 'student'

Next, you can try above example after removing comment from 'age' property as follows:

ApplicationContext.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for student bean -->

<bean id="student" class="com.tutorialspoint.Student">

<property name="name" value="Zara" />

<property name="age" value="11"/>

</bean>

</beans>

Now above example will produce following result:

Name : Zara

Age : 11

A more explicit example for @Required

<http://howtodoinjava.com/spring/spring-core/top-spring-core-interview-questions-with-answers/>

In a production-scale application, there may be hundreds or thousands of beans declared in the IoC container, and the dependencies between them are often very complicated. One of the shortcomings of setter injection is that it’s very hard for you to check if all required properties have been set or not. Spring’s dependency checking feature using “dependency-check” attribute, will not able to help you in this case. So solve this problem, you can use @Required annotation. After set the @Required annotation over setter method,

RequiredAnnotationBeanPostProcessor is a spring bean post processor that checks if all the bean properties with the @Required annotation have been set. To enable this bean post processor for property checking, you must register it in the Spring IoC container.

<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor" />

If any properties with @Required have not been set, a BeanInitializationException will be thrown by this bean post processor.

@Autowired

The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

@Autowired on Setter Methods:

You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

Here is the content of TextEditor.java file:

package com.tutorialspoint;

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

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker( ) {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file SpellChecker.java:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file Beans.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside SpellChecker constructor.

Inside checkSpelling.

@Autowired on Properties:

You can use @Autowired annotation on properties to get rid of the setter methods. When you will pass values of autowired properties using <property> Spring will automatically assign those properties with the passed values or references. So with the usage of @Autowired on properties your TextEditor.java file will become as follows:

package com.tutorialspoint;

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

public class TextEditor {

@Autowired

private SpellChecker spellChecker; 🡪 No set method for this property

public TextEditor() {

System.out.println("Inside TextEditor constructor." );

}

public SpellChecker getSpellChecker( ){

return spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Following is the configuration file Beans.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

Once you are done with the above two changes in source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside TextEditor constructor.

Inside SpellChecker constructor.

Inside checkSpelling.

@Autowired on Constructors:

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file. Let us check the following example.

Here is the content of TextEditor.java file:

package com.tutorialspoint;

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

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Following is the configuration file Beans.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/> 🡪 No construct-arg in this file.

<!-- Definition for textEditor bean without constructor-arg -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

Once you are done with the above two changes in source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside TextEditor constructor.

Inside SpellChecker constructor.

Inside checkSpelling.

@Qualifier

Situation you will encounter multiple beans of same type in spring

<http://stackoverflow.com/questions/18711924/required-multiple-beans-of-same-type-in-spring>

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

Here is the content of Student.java file:

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

Here is the content of Profile.java file:

package com.tutorialspoint;

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

import org.springframework.beans.factory.annotation.Qualifier;

public class Profile {

@Autowired

@Qualifier("student1")

private Student student;

public Profile(){

System.out.println("Inside Profile constructor." );

}

public void printAge() {

System.out.println("Age : " + student.getAge() );

}

public void printName() {

System.out.println("Name : " + student.getName() );

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

Profile profile = (Profile) context.getBean("profile");

profile.printAge();

profile.printName();

}

}

Consider the example of following configuration file Beans.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for profile bean -->

<bean id="profile" class="com.tutorialspoint.Profile">

</bean>

<!-- Definition for student1 bean -->

<bean id="student1" class="com.tutorialspoint.Student">

<property name="name" value="Zara" />

<property name="age" value="11"/>

</bean>

<!-- Definition for student2 bean -->

<bean id="student2" class="com.tutorialspoint.Student">

<property name="name" value="Nuha" />

<property name="age" value="2"/>

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside Profile constructor. 🡪 Only student1 which had @Qualifier will be autowired

Age : 11

Name : Zara

@JSR-250 series

<http://www.tutorialspoint.com/spring/spring_jsr250_annotations.htm>

1. Spring Java based configuration

Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations explained below.

@Configuration & @Bean Annotations:

Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The @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. The simplest possible @Configuration class would be as follows

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

Above code will be equivalent to the following XML configuration:

<beans>

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld" />

</beans>

Here the method name annotated with @Bean works as bean ID and it creates and returns actual bean. Your configuration class can have declaration for more than one @Bean. Once your configuration classes are defined, you can load & provide them to Spring container using AnnotationConfigApplicationContext as follows:

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

You can load various configuration classes as follows:

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx =

new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

E.g

Here is the content of HelloWorldConfig.java file:

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

Here is the content of HelloWorld.java file:

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

}

Once you are done with creating all the source files and adding required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, this will print the following message:

Your Message : Hello World!

Injecting Bean Dependency

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method calling another as follows.

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class AppConfig {

@Bean

public Foo foo() {

return new Foo(bar());

}

@Bean

public Bar bar() {

return new Bar();

}

}

Here, the foo bean receives a reference to bar via constructor injection

E.g

Here is the content of TextEditorConfig.java file:

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class TextEditorConfig {

@Bean

public TextEditor textEditor(){

return new TextEditor( spellChecker() );

}

@Bean

public SpellChecker spellChecker(){

return new SpellChecker( );

}

}

Here is the content of TextEditor.java file:

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file SpellChecker.java:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(TextEditorConfig.class);

TextEditor te = ctx.getBean(TextEditor.class);

te.spellCheck();

}

}

Once you are done with creating all the source filesand adding required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, this will print the following message:

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

The @Import annotation

The **@Import** annotation allows for loading @Bean definitions from another configuration class. Consider a ConfigA class as follows:

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

You can import above Bean declaration in another Bean Declaration as follows:

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B a() {

return new A();

}

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied as follows:

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

Lifecycle Callbacks:

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element:

public class Foo {

public void init() {

// initialization logic

}

public void cleanup() {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public Foo foo() {

return new Foo();

}

}

Specifying Bean Scope:

The default scope is singleton, but you can override this with the @Scope annotation as follows:

@Configuration

public class AppConfig {

@Bean

@Scope("prototype")

public Foo foo() {

return new Foo();

}

}

1. @Component, @Controller, @Repository & @Service annotations

<http://www.journaldev.com/2696/spring-interview-questions-and-answers>

<http://www.techferry.com/articles/spring-annotations.html>

<http://howtodoinjava.com/spring/spring-core/how-to-use-spring-component-repository-service-and-controller-annotations/>

@Component is used to indicate that a class is a component. These classes are used for auto detection and configured as bean, when annotation based configurations are used.

@Component

Annotate your other components (for example REST resource classes) with @Component.

E.g

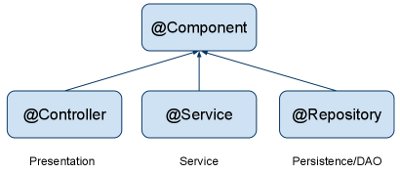
@Component

public class ContactResource {

...

}

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.



@Controller is a specific type of component, used in MVC applications and mostly used with RequestMapping annotation.

@Repository annotation is used to indicate that a component is used as repository and a mechanism to store/retrieve/search data. We can apply this annotation with DAO pattern implementation classes

E.g

Annotate all your DAO classes with @Repository. All your database access logic should be in DAO classes.

@Repository

public class CompanyDAOImpl implements CompanyDAO {

...

}

@Service is used to indicate that a class is a Service. Usually the business facade classes that provide some services are annotated with this.

E.g

Annotate all your service classes with @Service. All your business logic should be in Service classes.

@Service

public class CompanyServiceImpl implements CompanyService {

...

}

<http://www.techferry.com/articles/spring-annotations.html>

@RequestMapping

You use the @RequestMapping spring annotation to map URLs onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping.

@Controller

@RequestMapping("/company")

public class CompanyController {

@Autowired

private CompanyService companyService;

...

}

@PathVariable and @RequestParam

<http://stackoverflow.com/questions/19803731/spring-mvc-pathvariable>

suppose you want to write a url to fetch some order, you can say

www.mydomain.com/order/123

where 123 is orderId.

So now the url you will use in spring mvc controller would look like

/order/{orderId}

Now order id can be declared a path variable

@RequestMapping(value = " /order/{orderId}", method=RequestMethod.GET)

public String getOrder(@PathVariable String orderId){

//fetch order

}

if you use url www.mydomain.com/order/123, then orderId variable will be populated by value 123 by spring

Also note that PathVariable differ from requestParam as pathVariable are part of URL. The same url using request param would look like www.mydomain.com/order?orderId=123

@ModelAttribute

<http://stackoverflow.com/questions/3423262/what-is-modelattribute-in-spring-mvc>

|  |  |
| --- | --- |
|  | I know this is an old thread, but I thought I throw my hat in the ring and see if I can muddy the water a little bit more :)  I found my initial struggle to understand @ModelAttribute was a result of Spring's decision to combine several annotations into one. It became clearer once I split it into several smaller annotations:  For parameter annotations, think of @ModelAttribute as the equivalent of @Autowired + @Qualifieri.e. it tries to retrieve a bean with the given name from the Spring managed model. If the named bean is not found, instead of throwing an error or returning null, it implicitly takes on the role of @Bean i.e. Create a new instance using the default constructor and add the bean to the model.  For method annotations, think of @ModelAttribute as the equivalent of @Bean + @Before, i.e. it puts the bean constructed by user's code in the model and it's always called before a request handling method.  Figuratively, I see @ModelAttribute as the following (please don't take it literally!!):  @Bean("person")  @Before  public Person createPerson(){  return new Person();  }  @RequestMapping(...)  public xxx handlePersonRequest( (@Autowired @Qualifier("person") | @Bean("person")) Person person, xxx){  ...  }  As you can see, Spring made the right decision to make @ModelAttribute an all-encompassing annotation; no one wants to see an annotation smorgasbord. |

@SessionAttributes

@ModelAttributes and @SessionAttributes

<http://www.intertech.com/Blog/understanding-spring-mvc-model-and-session-attributes/>

<http://stackoverflow.com/questions/3106452/how-do-servlets-work-instantiation-shared-variables-and-multithreading/3106909#3106909>

SPRING MVC SCOPES

When I started writing Web applications in Spring MVC, I found the Spring model and session attributes to be a bit of a mystery – especially as they relate to the HTTP request and session scopes and their attributes that I knew well. Was a Spring model element going to be found in my session or request? If so, how could I control this? In this post, I hope to demystify how Spring MVC’s model and session work.

SPRING’S @MODELATTRIBUTE

There are several ways to add data or objects to Spring’s model. Data or objects are typically added to Spring’s model via an annotated method in the controller. In the example below, @ModelAttribute is used to add an instance of MyCommandBean to the model under the key of “myRequestObject”.

@CONTROLLER

PUBLIC CLASS MYCONTROLLER {

@MODELATTRIBUTE("MYREQUESTOBJECT")

PUBLIC MYCOMMANDBEAN ADDSTUFFTOREQUESTSCOPE() {

SYSTEM.OUT.PRINTLN("INSIDE OF ADDSTUFFTOREQUESTSCOPE");

MYCOMMANDBEAN BEAN = NEW MYCOMMANDBEAN("HELLO WORLD",42);

RETURN BEAN;

}

@REQUESTMAPPING("/DOSOMETHING")

PUBLIC STRING REQUESTHANDLINGMETHOD(MODEL MODEL, HTTPSERVLETREQUEST REQUEST) {

SYSTEM.OUT.PRINTLN("INSIDE OF DOSOMETHING HANDLER METHOD");

SYSTEM.OUT.PRINTLN("--- MODEL DATA ---");

MAP MODELMAP = MODEL.ASMAP();

FOR (OBJECT MODELKEY : MODELMAP.KEYSET()) {

OBJECT MODELVALUE = MODELMAP.GET(MODELKEY);

SYSTEM.OUT.PRINTLN(MODELKEY + " -- " + MODELVALUE);

}

SYSTEM.OUT.PRINTLN("=== REQUEST DATA ===");

JAVA.UTIL.ENUMERATION REQENUM = REQUEST.GETATTRIBUTENAMES();

WHILE (REQENUM.HASMOREELEMENTS()) {

STRING S = REQENUM.NEXTELEMENT();

SYSTEM.OUT.PRINTLN(S);

SYSTEM.OUT.PRINTLN("==" + REQUEST.GETATTRIBUTE(S));

}

RETURN "NEXTPAGE";

}

// ... THE REST OF THE CONTROLLER

}

On an incoming request, any methods annotated with @ModelAttribute are called before any controller handler method (like requestHandlingMethod in the example above). These methods add data to a java.util.Map that is added to the Spring model before the execution of the handler method. This can be demonstrated by a simple experiment. I created two JSP pages: index.jsp and nextpage.jsp. A link on index.jsp page is used to send a request into the application triggering the requestHandlingMethod() of MyController. Per the code above, the requestHandlingMethod() returns “nextpage” as the logical name of the next view which is resolved to nextpage.jsp in this case.

modeldataexample

When this little Web site is executed in this fashion, the System.out.println’s of the controller, show how the @ModelAttribute method is executed before the handler method. It also shows that the MyCommandBean was created and added to Spring’s model and was available in the handler method.

INSIDE OF ADDSTUFFTOREQUESTSCOPE

INSIDE OF DOSOMETHING HANDLER METHOD

--- MODEL DATA ---

MYREQUESTOBJECT -- MYCOMMANDBEAN [SOMESTRING=HELLO WORLD, SOMENUMBER=42]

=== REQUEST DATA ===

ORG.SPRINGFRAMEWORK.WEB.SERVLET.DISPATCHERSERVLET.THEME\_SOURCE

==WEBAPPLICATIONCONTEXT FOR NAMESPACE 'DISPATCHER-SERVLET': STARTUP DATE [SUN OCT 13 21:40:56 CDT 2013]; ROOT OF CONTEXT HIERARCHY

ORG.SPRINGFRAMEWORK.WEB.SERVLET.DISPATCHERSERVLET.THEME\_RESOLVER

==ORG.SPRINGFRAMEWORK.WEB.SERVLET.THEME.FIXEDTHEMERESOLVER@204AF48C

ORG.SPRINGFRAMEWORK.WEB.SERVLET.DISPATCHERSERVLET.CONTEXT

==WEBAPPLICATIONCONTEXT FOR NAMESPACE 'DISPATCHER-SERVLET': STARTUP DATE [SUN OCT 13 21:40:56 CDT 2013]; ROOT OF CONTEXT HIERARCHY

ORG.SPRINGFRAMEWORK.WEB.SERVLET.HANDLERMAPPING.PATHWITHINHANDLERMAPPING

==DOSOMETHING.REQUEST

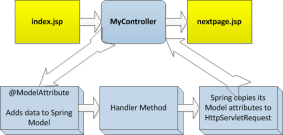
ORG.SPRINGFRAMEWORK.WEB.SERVLET.HANDLERMAPPING.BESTMATCHINGPATTERN

==/DOSOMETHING.\*

ORG.SPRINGFRAMEWORK.WEB.SERVLET.DISPATCHERSERVLET.LOCALE\_RESOLVER

==ORG.SPRINGFRAMEWORK.WEB.SERVLET.I18N.ACCEPTHEADERLOCALERESOLVER@18FD23E4

Now, the question is “where is Spring model data stored?” Is it stored in the standard Java request scope? The answer is – yes… eventually. As you can tell from the output above, MyCommandBean is in the model, but not yet in the request object when the handler method executes. Indeed, Spring does not add the model data to the request as an attribute until after the execution of the handler method and before presentation of the next view (in this case the nextpage.jsp).



This can also be demonstrated by printing out the attribute data stored in the HttpServletRequest in both index.jsp and nextpage.jsp. I arranged for both of these pages to use a JSP scriptlet (shown below) to display the HttpServletRequest attributes.

<HR />

<H3>REQUEST SCOPE (KEY==VALUES)</H3>

<%

JAVA.UTIL.ENUMERATION<STRING> REQENUM = REQUEST.GETATTRIBUTENAMES();

WHILE (REQENUM.HASMOREELEMENTS()) {

STRING S = REQENUM.NEXTELEMENT();

OUT.PRINT(S);

OUT.PRINTLN("==" + REQUEST.GETATTRIBUTE(S));

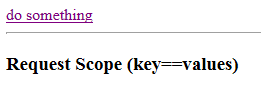
%><BR />

<%

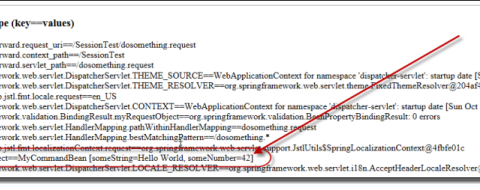
}

%>

When the application comes up and index.jsp is displayed, you can see that there are no attributes in request scope.



In this case, when the “do something” link is clicked it causes the MyController’s handler method to execute, which in turn causes the nextpage.jsp to be displayed. Given the same JSP scriptlet is on the nextpage.jsp, it too renders what is in the request scope. Lo and behold, when nextpage.jsp renders, it shows the model MyCommandBean created in the controller has been added to the HttpServletRequest scope! The Spring model attribute key of “myRequestObject” has even been copied and used as the request attribute’s key.



So Spring model data created prior to (or during) the handler method execution has been copied to the HttpServletRequest before the next view is rendered.

REASON BEHIND SPRING MODEL VERSUS REQUEST

You may wonder why Spring uses model attributes. Why not just add data directly to the request object? I found the answer to this question in Rod Johnson et. al’s book Professional Java Development with the Spring Framework. This book is a little dated with regard to the Spring API (written to Spring 2.0), but I have always found the text provide a little more explanation of what’s going on under the covers of the Spring engine. Here is the quote from that text regarding model elements:

“…adding elements directly to the HttpServletRequest (as request attributes) would seem to serve the same purpose. The reason to do this is obvious when taking a look at one of the requirements we have set for the MVC framework: It should be as view-agnostic as possible, which means we’d like to be able to incorporate view technologies not bound to the HttpServletRequest as well.” (page 429-430)

SPRING’S @SESSIONATTRIBUTES

So now you know how Spring’s model data is managed and how it relates to regular Http request attribute data. What about Spring’s session data?

Spring’s @SessionAttributes is used on a controller to designate which model attributes should be stored in the session. Actually, to be precise, the Spring documentation states that the @SessionAttributes annotation “list the names of model attributes which should be transparently stored in the session or some conversational storage.” Again, the “some conversational storage” suggests how Spring MVC tries to remain technology agnostic it is design.

In actually, what @SessionAttributes allows you to do is tell Spring which of your model attributes will also be copied to HttpSession before rendering the view. Again, this can be demonstrated with a little code.

In my index.jsp and nextpage.jsp, I added an additional JSP scriptlet to show the HttpSession attributes.

<H3>SESSION SCOPE (KEY==VALUES)</H3>

<%

JAVA.UTIL.ENUMERATION<STRING> SESSENUM = REQUEST.GETSESSION()

.GETATTRIBUTENAMES();

WHILE (SESSENUM.HASMOREELEMENTS()) {

STRING S = SESSENUM.NEXTELEMENT();

OUT.PRINT(S);

OUT.PRINTLN("==" + REQUEST.GETSESSION().GETATTRIBUTE(S));

%><BR />

<%

}

%>

I annotated MyController with @SessionAttributes to put the same model attribute (myRequestObject) in Spring session.

@CONTROLLER

@SESSIONATTRIBUTES("MYREQUESTOBJECT")

PUBLIC CLASS MYCONTROLLER {

...

}

I also added code to the handler method of my controller to show what attributes are in HttpSession (just as it shows what attributes are in HttpServletRequest).

@SUPPRESSWARNINGS("RAWTYPES")

@REQUESTMAPPING("/DOSOMETHING")

PUBLIC STRING REQUESTHANDLINGMETHOD(MODEL MODEL, HTTPSERVLETREQUEST REQUEST, HTTPSESSION SESSION) {

SYSTEM.OUT.PRINTLN("INSIDE OF DOSOMETHING HANDLER METHOD");

SYSTEM.OUT.PRINTLN("--- MODEL DATA ---");

MAP MODELMAP = MODEL.ASMAP();

FOR (OBJECT MODELKEY : MODELMAP.KEYSET()) {

OBJECT MODELVALUE = MODELMAP.GET(MODELKEY);

SYSTEM.OUT.PRINTLN(MODELKEY + " -- " + MODELVALUE);

}

SYSTEM.OUT.PRINTLN("=== REQUEST DATA ===");

JAVA.UTIL.ENUMERATION<STRING> REQENUM = REQUEST.GETATTRIBUTENAMES();

WHILE (REQENUM.HASMOREELEMENTS()) {

STRING S = REQENUM.NEXTELEMENT();

SYSTEM.OUT.PRINTLN(S);

SYSTEM.OUT.PRINTLN("==" + REQUEST.GETATTRIBUTE(S));

}

SYSTEM.OUT.PRINTLN("\*\*\* SESSION DATA \*\*\*");

ENUMERATION<STRING> E = SESSION.GETATTRIBUTENAMES();

WHILE (E.HASMOREELEMENTS()){

STRING S = E.NEXTELEMENT();

SYSTEM.OUT.PRINTLN(S);

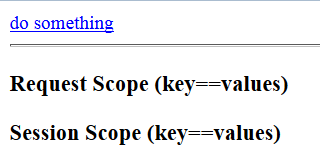
SYSTEM.OUT.PRINTLN("\*\*" + SESSION.GETATTRIBUTE(S));

}

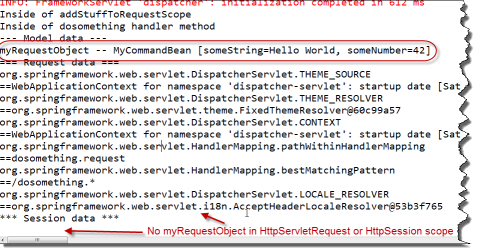
RETURN "NEXTPAGE";

}

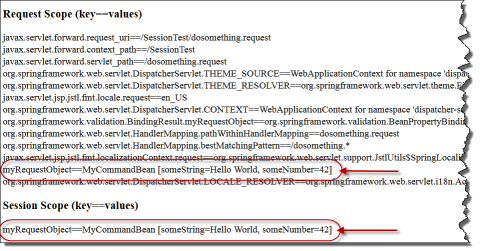
So now, we should be able to see what is in the session object before, during, and after Spring MVC has handled one HTTP request when annotated with @SessionAttributes. The results are shown below. First, as the index.jsp page is displayed (before the request is sent and handled by Spring MVC), we see that there is no attribute data in either the HttpServletRequest or HttpSession.



During the execution of the handler method (requestHandlingMethod), you see MyCommandBean has been added to the Spring model attributes, but it is not yet in the HttpServletRequest or HttpSession scope.



But after the handler method has executed and when the nextpage.jsp is rendered, you can see that the model attribute data (MyCommandBean) has indeed been copied as an attribute (with the same attribute key) to both HttpServletRequest and HttpSession.



CONTROLLING SESSION ATTRIBUTES

So now you have an appreciation of how Spring model and session attribute data are added to HttpServletRequest and HttpSession. But now you may be concerned with how to manage that data in Spring session. Spring provides a means to remove Spring session attributes, and thereby also remove it from HttpSession (without having to kill the entire HttpSession). Simply add a Spring SessionStatus object as a parameter to a controller handler method. In this method, use the SessionStatus object to end the Spring session.

@REQUESTMAPPING("/ENDSESSION")

PUBLIC STRING NEXTHANDLINGMETHOD2(SESSIONSTATUS STATUS){

STATUS.SETCOMPLETE();

RETURN "LASTPAGE";

}

WRAP UP

So hopefully, this post has helped you understand Spring model and session attributes. Its not magic, its just a matter of understanding how HttpSession and HttpServletRequest are used to store Spring model and session attributes.

1. Event handling in Spring

<http://www.tutorialspoint.com/spring/event_handling_in_spring.htm>

1. Custom event in Spring

Create event first

Here is the content of CustomEvent.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationEvent;

public class CustomEvent extends ApplicationEvent{

public CustomEvent(Object source) {

super(source);

}

public String toString(){

return "My Custom Event";

}

}

Create event publisher

Following is the content of the CustomEventPublisher.java file:

package com.tutorialspoint;

import org.springframework.context.ApplicationEventPublisher;

import org.springframework.context.ApplicationEventPublisherAware;

public class CustomEventPublisher

implements ApplicationEventPublisherAware {

private ApplicationEventPublisher publisher;

public void setApplicationEventPublisher

(ApplicationEventPublisher publisher){

this.publisher = publisher;

}

public void publish() {

CustomEvent ce = new CustomEvent(this);

publisher.publishEvent(ce);

}

}

Create event handler

Following is the content of the CustomEventHandler.java file.

package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

public class CustomEventHandler

implements ApplicationListener<CustomEvent>{

public void onApplicationEvent(CustomEvent event) {

System.out.println(event.toString());

}

}

Create main class

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ConfigurableApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

CustomEventPublisher cvp =

(CustomEventPublisher) context.getBean("customEventPublisher");

cvp.publish();

cvp.publish();

}

}

Create configuration file

Following is the configuration file Beans.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="customEventHandler"

class="com.tutorialspoint.CustomEventHandler"/>

<bean id="customEventPublisher"

class="com.tutorialspoint.CustomEventPublisher"/>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

y Custom Event

y Custom Event

1. Spring JDBC Framework overview

JDBCTemplate class

The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the JdbcTemplate class are threadsafe once configured. So you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs.

How JdbcTemplate can be used ?

A common practice when using the JdbcTemplate class is to configure a DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.

Configuring Data Source

Let us create a database table Student in our database TEST. I assume you are working with MySQL database, if you work with any other database then you can change your DDL and SQL queries accordingly.

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Now we need to supply a DataSource to the JdbcTemplate so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code as shown below:

<bean id="dataSource" 🡪 Note: Used for DataSource setting in DAO impl class.

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/TEST"/>

<property name="username" value="root"/>

<property name="password" value="password"/>

</bean>

Data Access Object (DAO)

DAO stands for data access object which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The Data Access Object (DAO) support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA or JDO in a consistent way.

E.g

<http://www.tutorialspoint.com/spring/spring_jdbc_example.htm>

Following is the content of the Data Access Object interface file StudentDAO.java:

package com.tutorialspoint;

import java.util.List;

import javax.sql.DataSource;

public interface StudentDAO {

/\*\*

\* This is the method to be used to initialize

\* database resources ie. connection.

\*/

public void setDataSource(DataSource ds);

/\*\*

\* This is the method to be used to create

\* a record in the Student table.

\*/

public void create(String name, Integer age);

/\*\*

\* This is the method to be used to list down

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public Student getStudent(Integer id);

/\*\*

\* This is the method to be used to list down

\* all the records from the Student table.

\*/

public List<Student> listStudents();

/\*\*

\* This is the method to be used to delete

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public void delete(Integer id);

/\*\*

\* This is the method to be used to update

\* a record into the Student table.

\*/

public void update(Integer id, Integer age);

}

Following is the content of the Student.java file:

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of the StudentMapper.java file:

Note: RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

<http://www.javatpoint.com/RowMapper-example>

package com.tutorialspoint;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setId(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Following is the implementation class file StudentJDBCTemplate.java for the defined DAO interface StudentDAO:

package com.tutorialspoint;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplateObject;

Note: If using other ORM structure like Hibernate, we should change DataSource / JdbcTemple to SessionFactory 🡪 Check Hibernate integration with Spring example.

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplateObject = new JdbcTemplate(dataSource);

} 🡪 Work with bean.xml properties ref setting to do bean dependency injection.

public void create(String name, Integer age) {

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, name, age); Note: Create = update with insert

System.out.println("Created Record Name = " + name + " Age = " + age);

return;

}

public Student getStudent(Integer id) {

String SQL = "select \* from Student where id = ?";

Student student = jdbcTemplateObject.queryForObject(SQL,

new Object[]{id}, new StudentMapper());

return student;

} Note: get = queryForObject with select \* from

public List<Student> listStudents() {

String SQL = "select \* from Student";

List <Student> students = jdbcTemplateObject.query(SQL,

new StudentMapper());

return students;

} Note: get Collections = query with RowMapper with select \* from

public void delete(Integer id){

String SQL = "delete from Student where id = ?";

jdbcTemplateObject.update(SQL, id);

System.out.println("Deleted Record with ID = " + id );

return;

} Note: delete = update with delete

public void update(Integer id, Integer age){

String SQL = "update Student set age = ? where id = ?";

jdbcTemplateObject.update(SQL, age, id);

System.out.println("Updated Record with ID = " + id );

return;

}

}

Following is the content of the MainApp.java file:

package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

StudentJDBCTemplate studentJDBCTemplate =

(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

System.out.println("------Records Creation--------" );

studentJDBCTemplate.create("Zara", 11);

studentJDBCTemplate.create("Nuha", 2);

studentJDBCTemplate.create("Ayan", 15);

System.out.println("------Listing Multiple Records--------" );

List<Student> students = studentJDBCTemplate.listStudents();

for (Student record : students) {

System.out.print("ID : " + record.getId() );

System.out.print(", Name : " + record.getName() );

System.out.println(", Age : " + record.getAge());

}

System.out.println("----Updating Record with ID = 2 -----" );

studentJDBCTemplate.update(2, 20);

System.out.println("----Listing Record with ID = 2 -----" );

Student student = studentJDBCTemplate.getStudent(2);

System.out.print("ID : " + student.getId() );

System.out.print(", Name : " + student.getName() );

System.out.println(", Age : " + student.getAge());

}

}

Following is the configuration file Beans.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

<!-- Initialization for data source -->

<bean id="dataSource" 🡪 Note: This is the DataSource bean name used for “ref”

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/TEST"/>

<property name="username" value="root"/>

<property name="password" value="password"/>

</bean>

<!-- Definition for studentJDBCTemplate bean -->

<bean id="studentJDBCTemplate" 🡪 Note:This is the DAO impl bean name

class="com.tutorialspoint.StudentJDBCTemplate">

<property name="dataSource" ref="dataSource" /> 🡪 Be careful, this is the critical part, use ref with setDataSource() method to do bean dependency injection.

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

------Records Creation--------

Created Record Name = Zara Age = 11

Created Record Name = Nuha Age = 2

Created Record Name = Ayan Age = 15

------Listing Multiple Records--------

ID : 1, Name : Zara, Age : 11

ID : 2, Name : Nuha, Age : 2

ID : 3, Name : Ayan, Age : 15

----Updating Record with ID = 2 -----

Updated Record with ID = 2

----Listing Record with ID = 2 -----

ID : 2, Name : Nuha, Age : 20

**Conclusion**

1. Structure: Beans.xml(applicaitonContext.xml) / DAO class / DAO impl class / RowMapper impl class.
2. Two critical beans definition in Beans.xml(applicaitonContext.xml):
3. DriverManagerDataSource 🡪 Used for matching “DataSource” field in DAO impl class. 🡪 The data source detail(here we are using MySql driver) is actually defined on xml file.
4. DAO impl class name which use “ref” to retrieve data source defined above.
5. Two critical fields defined in DAO impl class:
6. private DataSource with setDataSource() method to do DataSource DI.
7. private JdbcTemplate with new constructor to create inside setDataSource()
8. DAO impl class should contain methods as:

JdbcTemplate is used for communicate with data base to do real function, very similar to Session in Hibernate, here, jdbcTemplateObject is instance of JdbcTemplate.

1. Create = jdbcTemplateObject.update() with insert query.
2. queryForObject = jdbcTemplateObject.queryForObject() with select query
3. queryForObject collection = jdbcTemplateObject.query() with RowMapper
4. Delete = jdbcTemplateObject.update() with delete query.
5. For query collections result from ResultSet, RowMapper vs ResultSetExtractor

<http://stackoverflow.com/questions/10074025/what-is-difference-between-resultsetextractor-vs-rowmapper>

1. How to call stored procedure in Spring ?

<http://javarevisited.blogspot.com/2013/04/spring-framework-tutorial-call-stored-procedures-from-java.html>

Spring Framework provides excellent support to call stored procedures from Java application. In fact there are multiple ways to call stored procedure in Spring Framework, e.g. you can use one of the query() method from JdbcTemplate to call stored procedures, or you can extend [abstract class](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html) StoredProcedure to call stored procedures from Java. In this Java Spring tutorial, we will see second approach to call stored procedure. It's more [object oriented](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html), but same time requires more coding. StoredProcedure class allows you to declare IN and OUT parameters and call stored procedure using its various execute() method, which has protected access and can only be called from sub class. I personally prefer to implement StoredProcedure class as [Inner class](http://javarevisited.blogspot.sg/2012/12/inner-class-and-nested-static-class-in-java-difference.html), if its tied up with one of [DAO Object](http://javarevisited.blogspot.com/2013/01/data-access-object-dao-design-pattern-java-tutorial-example.html), e.g. in this case it nicely fit inside EmployeeDAO. Then you can provide convenient method to wrap stored procedure calls. In order to demonstrate, how to call stored procedures from spring based application, we will first create a simple stored proc using MySQL database, as shown below.

MySQL Stored procedure

We will use following stored procedure for this example. This is created in MySQL database and accept an input parameter IN, which is employeeId and return name of employee using its output parameter called, name.

mysql> DELIMITER //

mysql> create procedure usp\_GetEmployeeName(IN id INT, OUT name VARCHAR(20))

-> begin

-> select emp\_name into name from employee where emp\_id = id;

-> end//

Query OK, 0 rows affected (0.52 sec)

mysql> DELIMITER ;

For quick test, you can also call this stored procedure in mysql, assuming you have employee table as discussed in this article and some data on it. To learn more about stored proc in MySQL, see [How to create and call MySQL stored procedure form command line](http://javarevisited.blogspot.com/2013/02/-create-and-call-mysql-stored-procedure-database-sql-example-tutorial.html).

mysql> call usp\_GetEmployeeName(103, @name);

Query OK, 1 row affected (0.05 sec)

mysql> select @name;

+*-------+*

| @name |

+*-------+*

| Jack |

+*-------+*

1 row in set (0.00 sec)

Here is complete code example of how to call stored procedure from Spring framework. In this example, we have extended [abstract class](http://java67.blogspot.com/2013/02/can-abstract-class-have-constructor-in-java.html) StoredProcedure in our class called, EmployeeSP. This is declared as [nested class](http://java67.blogspot.com/2012/10/nested-class-java-static-vs-non-static-inner.html) inside EmployeeDAO because its only used by this class, if your stored procedure is used my multiple DAO classes, than you can also make it a top level class. If you look at [constructor](http://javarevisited.blogspot.com/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) of EmployeeSP, it calls super class constructor and passes datasource and name of database stored procedure. We have also declared two stored procedure parameters, one is IN parameter id, and other is OUT parameter. Input to stored procedure is passed using IN parameter, and output from stored procedure is read using OUT parameter. Your stored procedure can have multiple IN and OUT parameter. StoredProcedure class also provide several execute() methods, which can be invoked to call stored procedure and get result. It return result as [Map](http://javarevisited.blogspot.com/2011/12/how-to-traverse-or-loop-hashmap-in-java.html), where key is OUT parameter, and value is result of stored procedure. Here is the code for DAO class and stored procedure along with Spring Configuration file, since Spring framework is based on principle of [dependency Injection and Inversion of control](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html), this file is required to create and manage object.

Java Class which wraps Stored procedure

import java.sql.Types;

import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.SqlOutParameter;

import org.springframework.jdbc.core.SqlParameter;

import org.springframework.jdbc.object.StoredProcedure;

public class EmployeeDao {

private JdbcTemplate jdbcTemplate;

private EmployeeSP sproc;

public void setDataSource(DataSource source){

this.jdbcTemplate = new JdbcTemplate(source);

this.sproc = new EmployeeSP(jdbcTemplate.getDataSource());

}

*/\**

*\* wraps stored procedure call*

*\*/*

public String getEmployeeName(int emp\_id){

return (String) sproc.execute(emp\_id);

}

*/\**

*\* Inner class to implement stored procedure in spring.*

*\*/*

private class EmployeeSP extends *StoredProcedure*{

private static final String SPROC\_NAME = "usp\_GetEmployeeName";

public EmployeeSP( DataSource datasource ){

super( datasource, SPROC\_NAME );

declareParameter( new SqlParameter( "id", Types.INTEGER) ); *//declaring sql in parameter to pass input*

declareParameter( new SqlOutParameter( "name", Types.VARCHAR ) ); *//declaring sql out parameter*

compile();

}

public Object execute(int emp\_id){

Map<String,Object> results = super.execute(emp\_id);

return results.get("name"); *//reading output of stored procedure using out parameters*

}

}

}

Main class to test stored procedure

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

*/\**

*\* Main class to start and test this Java application*

*\*/*

public class Main {

public static void main(String args[]){

ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config.xml");

EmployeeDao dao = (EmployeeDao) ctx.getBean("employeeDao");

*//calling stored procedure using DAO method*

System.out.println("Employee name for id 103 is : " + dao.getEmployeeName(103));

}

}

Output

2013-01-17 23:56:34,408 0 [main] DEBUG EmployeeDao$EmployeeSP - Compiled stored procedure. Call string is [{call usp\_GetEmployeeName(?, ?)}]

2013-01-17 23:56:34,439 31 [main] DEBUG EmployeeDao$EmployeeSP - RdbmsOperation with SQL [usp\_GetEmployeeName] compiled

Employee name for id 103 is : Jack

Spring configuration file:

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

<beans xmlns="http:**//**www.springframework.org/schema/beans"

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

        xmlns:jms="http:**//**www.springframework.org/schema/jms" xmlns:context="http:**//**www.springframework.org/schema/context"

        xsi:schemaLocation="

http:**//**www.springframework.org/schema/beans http:**//**www.springframework.org/schema/beans/spring-beans-3.0.xsd

http:**//**www.springframework.org/schema/jms http:**//**www.springframework.org/schema/jms/spring-jms-3.0.xsd

http:**//**www.springframework.org/schema/context http:**//**www.springframework.org/schema/context/spring-context-3.0.xsd">

        <bean id="propertyPlaceholderConfigurer"

                class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

                <property name="locations">

                        <list>

                                <value>classpath:jdbc.properties</value>

                        </list>

                </property>

        </bean>

    <bean id="springDataSource" class="org.springframework.jdbc.datasource.SingleConnectionDataSource">

        <property name="driverClassName" value="${db.driver}" />

        <property name="url" value="${db.url}" />

        <property name="username" value="root" />

        <property name="password" value="root" />

</bean>

        <bean id="employeeDao" class ="EmployeeDao">

                <property name="dataSource" ref="springDataSource"/>

        </bean>

</beans>

1. How to setup JDBC connection pool in Spring ?

<http://javarevisited.blogspot.com/2012/06/jdbc-database-connection-pool-in-spring.html>

**Spring framework** provides convenient JdbcTemplate class for performing all Database related operation. if you are not using Hibernate than using Spring's JdbcTemplate is good option. JdbcTemplate requires a DataSource which is javax.sql.DataSource implementation and you can get this directly using [spring bean](http://javarevisited.blogspot.sg/2012/05/what-is-bean-scope-in-spring-mvc.html) configuration or by using **JNDI** if you are using [J2EE web server or application server](http://javarevisited.blogspot.sg/2012/05/5-difference-between-application-server.html) for managing Connection Pool. See *How to setup JDBC connection Pool in tomcat and Spring* for JNDI based connection pooling for more details. In order to setup Data source you will require following configuration in your applicationContext.xml (spring configuration) file:

//Datasource connection settings in Spring  
**<bean** id="springDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close" **>**  
   **<property** name="url" value="jdbc:oracle:thin:@localhost:1521:SPRING\_TEST" **/>**  
   **<property** name="driverClassName" value="oracle.jdbc.driver.OracleDriver" **/>**  
   **<property** name="username" value="root" **/>**  
   **<property** name="password" value="root" **/>**  
   **<property** name="removeAbandoned" value="true"**/>**  
   **<property** name="initialSize" value="20" **/>**  
   **<property** name="maxActive" value="30" **/>**  
**</bean>**  
  
//Dao class configuration in spring  
 **<bean** id="EmployeeDatabaseBean" class="com.test.EmployeeDAOImpl"**>**  
    **<property** name="dataSource" ref="springDataSource"**/>**  
 **</bean>**  
  
Below configuration of DBCP connection pool will create 20 database connection as initialSize is 20 and goes up to 30 Database connection if required as maxActive is 30. you can customize your database connection pool by using different properties provided by Apache DBCP library. Above example is creating connection pool with Oracle 11g database and we are using oracle.jdbc.driver.OracleDriver comes along with **ojdbc6.jar or ojdbc6\_g.jar ,** to learn more about [how to connect Oracle database from Java program](http://javarevisited.blogspot.sg/2012/04/java-program-to-connect-oracle-database.html) see the link.

Below is **complete code example of DAO class which uses Spring JdbcTemplate** to execute SELECT query against database using database connection from Connection pool. If you are not initializing Database connection pool on start-up than it may take a while when you execute your first query because it needs to create certain number of SQL connection and then it execute query but once connection pool is created subsequent queries will execute faster.

*//Code for DAO Class using Spring JdbcTemplate*  
**package** com.test  
**import** javax.sql.DataSource;  
**import** org.log4j.Logger;  
**import** org.log4j.LoggerFactory;  
**import** org.springframework.jdbc.core.JdbcTemplate;  
  
***/\*\*  
 \* Java Program example to use DBCP connection pool with Spring framework  
 \* @author Javin Paul  
 \*/***  
**public** **class** EmployeeDAOImpl **implements** EmployeeDAO {  
  
    **private** **Logger** logger = LoggerFactory.getLogger(EmployeeDAOImpl.**class**);  
    **private** JdbcTemplate jdbcTemplate;  
  
    **public** **void** setDataSource([**DataSource**](http://java.sun.com/j2se/1.5.0/docs/api/javax/sql/DataSource.html) dataSource) {  
        **this**.jdbcTemplate = **new** JdbcTemplate(dataSource);  
    }  
  
    @**Override**  
    **public** **boolean** isEmployeeExists(**String** emp\_id) {  
        **try** {  
            logger.debug("Checking Employee in EMP table using Spring Jdbc Template");  
            **int** number = **this**.jdbcTemplate.queryForInt("select count(\*) from EMP where emp\_id=?", emp\_id);  
            **if** (number > 0) {  
                **return** **true**;  
            }  
        } **catch** (**Exception** exception) {  
            exception.printStackTrace();  
        }  
        **return** **false**;  
    }  
}

1. Spring MVC framework

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

The Model encapsulates the application data and in general they will consist of POJO.

The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.

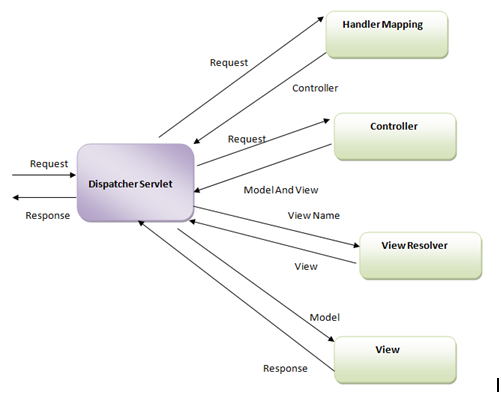
The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram:



<http://www.wideskills.com/spring/spring-mvc-framework>



Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

(1)After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.

(2)The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.

(3)The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.

(4)Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

All the above mentioned components ie. HandlerMapping, Controller and ViewResolver are parts of WebApplicationContext which is an extension of the plain ApplicationContext with some extra features necessary for web applications.

WebApplicationContext

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that it knows which servlet it is associated with.

Required Configuration

ou need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the web.xml file. The following is an example to show declaration and mapping for HelloWeb DispatcherServlet example:

<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">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>\*.jsp</url-pattern>

</servlet-mapping>

</web-app>

The web.xml file will be kept WebContent/WEB-INF directory of your web application. OK, upon initialization of HelloWeb DispatcherServlet, the framework will try to load the application context from a file named [servlet-name]-servlet.xml located in the application's WebContent/WEB-INF directory. In this case our file will be HelloWeb-servlet.xml.

Next, <servlet-mapping> tag indicates what URLs will be handled by the which DispatcherServlet. Here all the HTTP requests ending with .jsp will be handled by the HelloWeb DispatcherServlet.

Now, let us check the required configuration for HelloWeb-servlet.xml file, placed in your web application's WebContent/WEB-INF directory:

<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-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

Following are the important points about HelloWeb-servlet.xml file:

The [servlet-name]-servlet.xml file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.

The <context:component-scan...> tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.

The InternalResourceViewResolver will have rules defined to resolve the view names. As per the above defined rule, a logical view named hello is delegated to a view implementation located at /WEB-INF/jsp/hello.jsp .

Define a controller

DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The @Controller annotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

@Controller

@RequestMapping("/hello")

public

class HelloController{

@RequestMapping(method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The @Controller annotation defines the class as a Spring MVC controller. Here, the first usage of @RequestMapping indicates that all handling methods on this controller are relative to the /hello path. Next annotation @RequestMapping(method = RequestMethod.GET) is used to declare the printHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write above controller in another form where you can add additional attributes in @RequestMapping as follows:

@Controller

public class HelloController{

@RequestMapping(value = "/hello", method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The value attribute indicates the URL to which the handler method is mapped and the method attribute defines the service method to handle HTTP GET request. There are following important points to be noted about the controller defined above:

You will defined required business logic inside a service method. You can call another methods inside this method as per requirement.

Based on the business logic defined, you will create a model within this method. You can setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".

A defined service method can return a String which contains the name of the view to be used to render the model. This example returns "hello" as logical view name.

Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports etc. But most commonly we use JSP templates written with JSTL. So let us write a simple hello view in /WEB-INF/hello/hello.jsp:

<html>

<head>

<title>Hello Spring MVC</title>

</head>

<body>

<h2>${message}</h2> 🡪 Hello Spring MVC Framework!

</body>

</html>

Here ${message} is the attribute which we have setup inside the Controller. You can have multiple attributes to be displayed inside your view.

E.g Spring helloworld example

<http://www.tutorialspoint.com/spring/spring_mvc_hello_world_example.htm>

web.xml

<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">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Spring Web configuration file HelloWeb-servlet.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-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

Hello.jsp

<%@ page contentType="text/html; charset=UTF-8" %>

<html>

<head>

<title>Hello World</title>

</head>

<body>

<h2>${message}</h2>

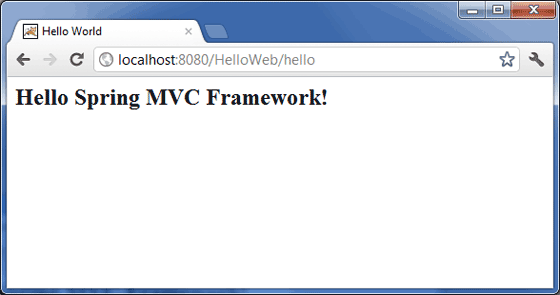
</body>

</html>

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in WebContent/WEB-INF/lib folder.

Once you are done with creating source and configuration files, export your application. Right click on your application and use Export > WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try to access the URL http://localhost:8080/HelloWeb/hello and if everything is fine with your Spring Web Application, you should see the following result:



You should note that in the given URL, HelloWeb is the application name and hello is the virtual subfolder which we have mentioned in our controller using @RequestMapping("/hello"). You can use direct root while mapping your URL using @RequestMapping("/"), in this case you can access the same page using short URL http://localhost:8080/HelloWeb/ but it is advised to have different functionalities under different folders.

E.g Spring MVC form handling

<http://www.tutorialspoint.com/spring/spring_mvc_form_handling_example.htm>

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Create a *Dynamic Web Project* with a name *HelloWeb* and create a package *com.tutorialspoint* under the *src* folder in the created project. |
| 2 | Drag and drop below mentioned Spring and other libraries into the folder*WebContent/WEB-INF/lib*. |
| 3 | Create a Java classes *Student* and *StudentController* under the*com.tutorialspoint* package. |
| 4 | Create Spring configuration files *Web.xml* and *HelloWeb-servlet.xml* under the *WebContent/WEB-INF* folder. |
| 5 | Create a sub-folder with a name *jsp* under the *WebContent/WEB-INF* folder. Create a view files *student.jsp* and *result.jsp* under this sub-folder. |
| 6 | The final step is to create the content of all the source and configuration files and export the application as explained below. |

Here is the content of Student.java file:

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of StudentController.java file:

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.ui.ModelMap;

@Controller

public class StudentController {

@RequestMapping(value = "/student", method = RequestMethod.GET)

public ModelAndView student() {

return new ModelAndView("student", "command", new Student());

}

@RequestMapping(value = "/addStudent", method = RequestMethod.POST)

public String addStudent(@ModelAttribute("SpringWeb")Student student,

ModelMap model) {

model.addAttribute("name", student.getName());

model.addAttribute("age", student.getAge());

model.addAttribute("id", student.getId());

return "result";

}

}

Here the first service method student(), we have passed a blank Student object in the ModelAndView object with name "command" because the spring framework expects an object with name "command" if you are using <form:form> tags in your JSP file. So when student() method is called it returns student.jsp view.

Second service method addStudent() will be called against a POST method on the HelloWeb/addStudent URL. You will prepare your model object based on the submitted information. Finally a "result" view will be returned from the service method, which will result in rendering result.jsp

Following is the content of Spring Web configuration file web.xml

<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">

<display-name>Spring MVC Form Handling</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file HelloWeb-servlet.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-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

Following is the content of Spring view file student.jsp

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Student Information</h2>

<form:form method="POST" action="/HelloWeb/addStudent">

<table>

<tr>

<td><form:label path="name">Name</form:label></td>

<td><form:input path="name" /></td>

</tr>

<tr>

<td><form:label path="age">Age</form:label></td>

<td><form:input path="age" /></td>

</tr>

<tr>

<td><form:label path="id">id</form:label></td>

<td><form:input path="id" /></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Submit"/>

</td>

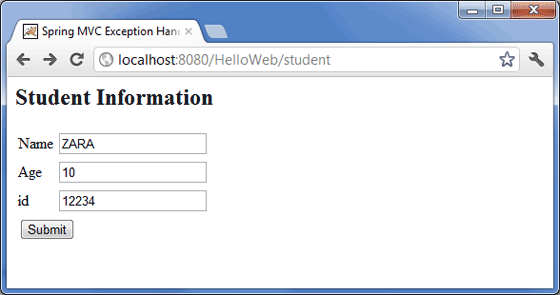
</tr>

</table>

</form:form>

</body>

</html>



Following is the content of Spring view file result.jsp

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Submitted Student Information</h2>

<table>

<tr>

<td>Name</td>

<td>${name}</td>

</tr>

<tr>

<td>Age</td>

<td>${age}</td>

</tr>

<tr>

<td>ID</td>

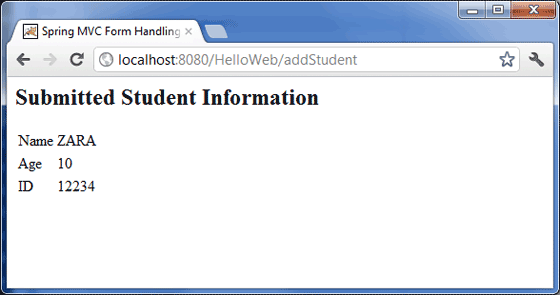
<td>${id}</td>

</tr>

</table>

</body>

</html>



1. What is difference between MVC pattern and 3-tier architecture ?

<http://programmers.stackexchange.com/questions/175950/in-mvc-dao-should-be-called-from-controller-or-model>

In my opinion, you have to distinguish between the MVC pattern and the 3-tier architecture. To sum up:

3-tier architecture:

data: persisted data; 🡪 DAO implementation with @Repository

service: logical part of the application; 🡪 Service implementation with @Service

presentation: hmi, webservice...

The MVC pattern takes place in the presentation tier of the above architecture (for a webapp):

data: ...;

service: ...;

presentation:

controller: intercepts the HTTP request and returns the HTTP response;

model: stores data to be displayed/treated;

view: organize output/display.

Life cycle of a typical HTTP request: 🡪 Note: Critical answer in interview

(1)The user sends the HTTP request;

(2)The controller intercepts it;

(3)The controller calls the appropriate service;

(4)The service calls the appropriate dao, which returns some persisted data (for example);

(5)The service treats the data, and returns data to the controller;

(6)The controller stores the data in the appropriate model and calls the appropriate view;

(7)The view get instantiated with the model's data, and get returned as the HTTP response.

Following is the best example for above 7 steps

<http://howtodoinjava.com/spring/spring-mvc/spring-mvc-hello-world-example/>

<http://www.slideshare.net/guestd0cc01/3-tier-architecture>

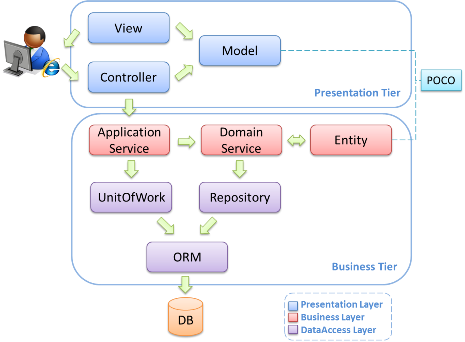
What is MVC Framework?

Model-view-controller (MVC) is a well known design pattern for designing UI based applications. It mainly decouples business logic from UIs by separating the roles of model, view, and controller in an application. Usually, models are responsible for encapsulating application data for views to present. Views should only present this data, without including any business logic. And controllers are responsible for receiving requests from users and invoking back-end services (manager or dao) for business logic processing. After processing, back-end services may return some data for views to present. Controllers collect this data and prepare models for views to present. The core idea of the MVC pattern is to separate business logic from UIs to allow them to change independently without affecting each other.



In a Spring MVC application, models usually consist of POJO objects that are processed by the service layer and persisted by the persistence layer. Views are usually JSP templates written with Java Standard Tag Library (JSTL). Controller part is played by dispatcher servlet which we will learn about in this tutorial in more detail.

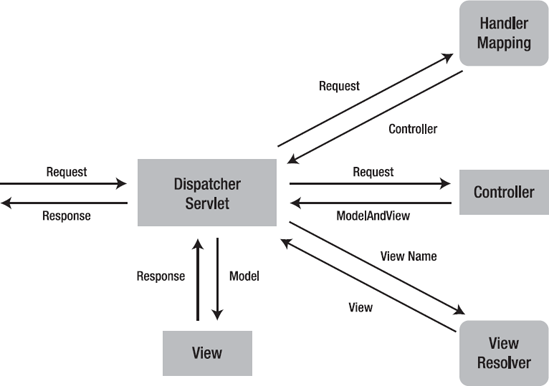
Some developers consider the service layer and DAO layers classes as part of model component in MVC. I have a different opinion on this. I do not consider service and DAO layers classes the part of MVC framework. Usually a web application is 3-tier architecture i.e. data-service-presentation. MVC is actually part of presentation layer



Dispatcher Servlet (Spring Controller)

In the simplest Spring MVC application, a controller is the only servlet you need to configure in a Java web deployment descriptor (i.e., the web.xml file). A Spring MVC controller—often referred to as a Dispatcher Servlet implements front controller design pattern and every web request must go through it so that it can manage the entire request life cycle.

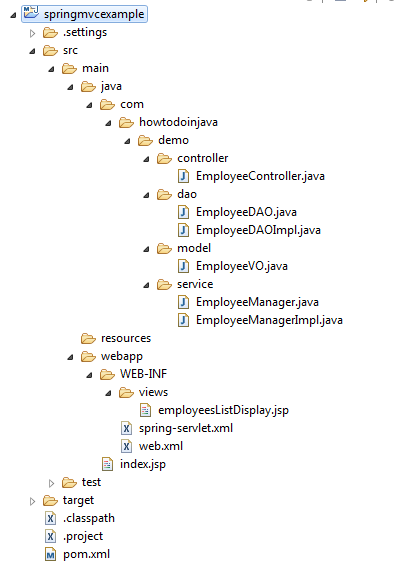
When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring’s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.



To define a controller class in Spring 3.0, a class has to be marked with the @Controller annotation. When a @Controller annotated controller receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class’s methods are decorated with the @RequestMapping annotation, making them handler methods.

After a handler method has finished processing the request, it delegates control to a view, which is represented as handler method’s return value. To provide a flexible approach, a handler method’s return value doesn’t represent a view’s implementation but rather a logical view i.e. without any file extension. You can map these logical views to right implementation into applicationContext file so that you can easily change your view layer code without even touching request handler class code.

To resolve the correct file for a logical name is the responsibility of view resolvers. Once the controller class has resolved a view name into a view implementation, per the view implementation’s design, it renders the objects.



Spring MVC Hello World Example

In this application, I am creating most simple employee management application demo having only one feature i.e. list all available employees in system. Let’s note down the directory structure of this application.

pom.xml

Below pom.xml file contains dependencies for spring mvc and taglibs support for writing jsp files.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.howtodoinjava.demo</groupId>

<artifactId>springmvcexample</artifactId>

<packaging>war</packaging>

<version>1.0-SNAPSHOT</version>

<name>springmvcexample Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!-- Spring MVC support -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.1.4.RELEASE</version>

</dependency>

<!-- Tag libs support for view layer -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId> 🡪 Note: Here is relate to Java Stand Tag Library

<version>1.2</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>taglibs</groupId>

<artifactId>standard</artifactId>

<version>1.1.2</version>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<finalName>springmvcexample</finalName>

</build>

</project>

web.xml

This minimum web.xml file declares one servlet (i.e. dispatcher servlet) to receive all kind of requests. Dispatcher servlet here acts as front controller.

<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">

<display-name>Spring Web MVC Hello World Application</display-name>

Note: Here in web.xml we define servlet as DispatchServlet and mapping for url handle

<servlet>

<servlet-name>spring</servlet-name> 🡪 Note: Servlet name is “spring” will match for servletname-servlet.xml below which for loading application context

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

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

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app

spring-servlet.xml (You can have applicationContext.xml as well)

We are using annotated classes at request handler, service and dao layer so I have enabled annotation processing for all class files in base package “com.howtodoinjava.demo“. The servletname-servlet.xml used for loading application context when initialize DispatchServlet.

See <http://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm>

<beans xmlns="http://www.springframework.org/schema/beans"

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

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/views/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

Note: What is InternalResourceViewResolver ?

<http://www.mkyong.com/spring-mvc/spring-mvc-internalresourceviewresolver-example/>

In Spring MVC, InternalResourceViewResolver is used to resolve “internal resource view” (in simple, it’s final output, jsp or htmp page) based on a predefined URL pattern. In additional, it allow you to add some predefined prefix or suffix to the view name (prefix + view name + suffix), and generate the final view page URL.

What’s internal resource views?

In Spring MVC or any web application, for good practice, it’s always recommended to put the entire views or JSP files under “WEB-INF” folder, to protect it from direct access via manual entered URL. Those views under “WEB-INF” folder are named as internal resource views, as it’s only accessible by the servlet or Spring’s controllers class.

EmployeeController.java

Annotation @RequestMapping at class level and method level determine the URL at which method will be invoked.

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import com.howtodoinjava.demo.service.EmployeeManager;

@Controller

@RequestMapping("/employee-module")

public class EmployeeController

{

**@Autowired**

**EmployeeManager manager; 🡪 Note: Controller contain Service bean**

@RequestMapping(value = "/getAllEmployees", method = RequestMethod.GET) 🡪

Note: Controller work with requestmapping

public String getAllEmployees(Model model) 🡪 Note: Controller operates with model

{

model.addAttribute("employees", manager.getAllEmployees());

🡪 Note: This “employee” attribute here will mapping to the employeesListDisplay.jsp’s definition as <c:forEach items="${employees}" var="employee">, the usage can be check on

<http://stackoverflow.com/questions/18975077/how-to-add-object-in-using-model-addattributes-in-spring-mvc>

<http://stackoverflow.com/questions/20584411/passing-a-new-object-in-spring-mvc-model-addattribute-method>

return "employeesListDisplay"; 🡪 Note: return with string pattern as logical view without file extension.

}

}

EmployeeVO.java

This class act as model for MVC pattern.-->

Note: Usually, models are responsible for encapsulating application data for views to present.

package com.howtodoinjava.demo.model;

import java.io.Serializable;

public class EmployeeVO implements Serializable

{

private static final long serialVersionUID = 1L;

private Integer id;

private String firstName;

private String lastName;

//Setters and Getters

@Override

public String toString() {

return "EmployeeVO [id=" + id + ", firstName=" + firstName

+ ", lastName=" + lastName + "]";

}

}

EmployeeDAO.java

The classes at third tier in 3-tier architecture. Responsible for interacting with underlying DB storage. 🡪 Data layer

import java.util.List;

import com.howtodoinjava.demo.model.EmployeeVO;

public interface EmployeeDAO

{

public List<EmployeeVO> getAllEmployees();

}

EmployeeDAOImpl.java

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Repository;

import com.howtodoinjava.demo.model.EmployeeVO;

@Repository 🡪 Note: The data layer class should note with @Repository

public class EmployeeDAOImpl implements EmployeeDAO {

public List<EmployeeVO> getAllEmployees()

{

List<EmployeeVO> employees = new ArrayList<EmployeeVO>();

EmployeeVO vo1 = new EmployeeVO();

vo1.setId(1);

vo1.setFirstName("Lokesh");

vo1.setLastName("Gupta");

employees.add(vo1);

EmployeeVO vo2 = new EmployeeVO();

vo2.setId(2);

vo2.setFirstName("Raj");

vo2.setLastName("Kishore");

employees.add(vo2);

return employees;

}

}

EmployeeManager.java

The classes at second tier in 3-tier architecture. Responsible for interacting with DAO Layer. 🡪 Service layer

import java.util.List;

import com.howtodoinjava.demo.model.EmployeeVO;

public interface EmployeeManager

{

public List<EmployeeVO> getAllEmployees();

}

EmployeeManagerImpl.java

import java.util.List;

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

import org.springframework.stereotype.Service;

import com.howtodoinjava.demo.dao.EmployeeDAO;

import com.howtodoinjava.demo.model.EmployeeVO;

@Service

public class EmployeeManagerImpl implements EmployeeManager {

**@Autowired**

**EmployeeDAO dao; 🡪 Note: Service layer class contain Data layer bean**

public List<EmployeeVO> getAllEmployees()

{

return dao.getAllEmployees();

}

}

employeesListDisplay.jsp

This jsp is used to display all the employees in system. It iterates the collection of employees in loop, and print their details in a table. This fits into view layer of MVC pattern.

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<html>

<head>

<title>Spring MVC Hello World</title>

</head>

<body>

<h2>All Employees in System</h2>

<table border="1">

<tr>

<th>Employee Id</th>

<th>First Name</th>

<th>Last Name</th>

</tr>

<c:forEach items="${employees}" var="employee"> 🡪 Note: The ${employee} name match the “attributeName” in model.addAttribute(attributeName, serviceImplClass.method())

<tr>

<td>${employee.id}</td>

<td>${employee.firstName}</td>

<td>${employee.lastName}</td>

</tr>

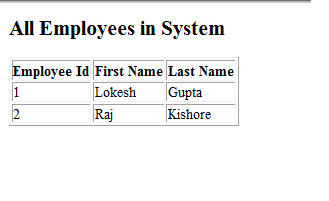
</c:forEach>

</table>

</body>

</html>

Now deploy the application in your application server (i am using tomcat 7). And hit the URL “http://localhost:8080/springmvcexample/employee-module/getAllEmployees“. You will see below screen if you have configured everything correctly.



Conclusion

Web.xml 🡪 Servlet/Servlet-mapping 🡪 DispatchServlet 🡪 ServletName-servlet.xml 🡪 InternalResourceViewResolver

Controller (@Controller) 🡪 RequestMapping (@RequestMapping) 🡪 value/method 🡪 model/ModelAndView/modelAttribute 🡪 JSP

Presentation layer (Spring MVC) 🡪 Businesss layer (Service/@Service) 🡪 Data Access layer (DAO impl/@Repository) 🡪 @Autowired

Combine data/service/presentation 3-tier together with how jsp used with Spring and how hibernate integrate with Spring (Not through JDBCtemplate but SessionFactory initialized in DAO(data layer interface) implement class), this is a full picture of CYLC project Spring design pattern.

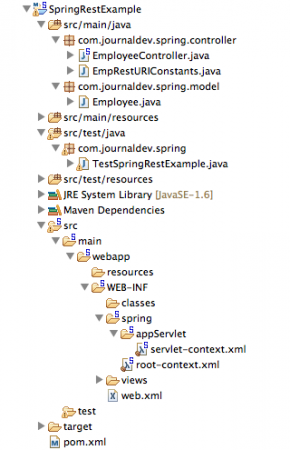
(Note: If using Java based annotation, we should use @Configuration + @Bean + @Import)

1. Spring Restful Web Services Example with JSON, Jackson and Client Program

<http://www.journaldev.com/2552/spring-rest-example-tutorial-spring-restful-web-services>

We will learn to create **Spring Restful Web Services** using **Spring MVC** and then test it out with the Rest client. At the end, we will also look into how to invoke Spring Restful web service using **Spring RestTemplate API**.

Create a new Spring MVC Project in the STS, our final project will look like below image. We will look into each of the components one by one.



Spring REST Configuration XML Files

pom.xml



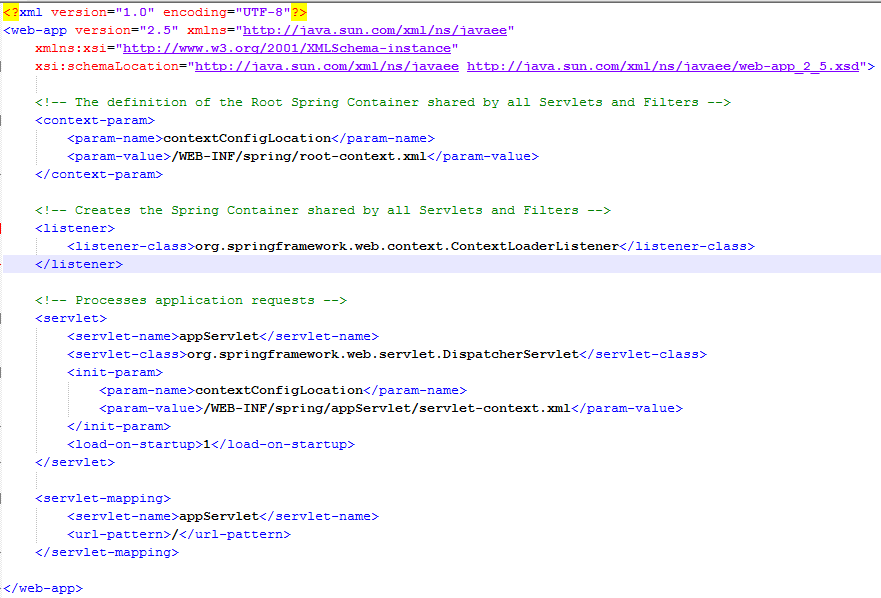




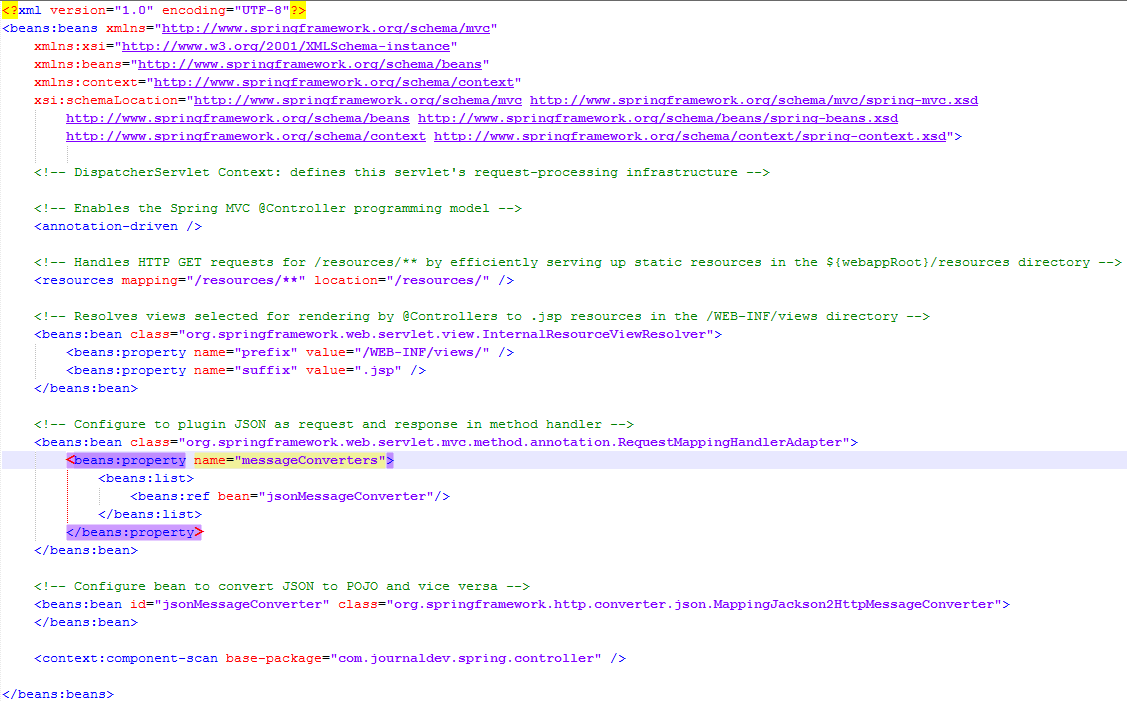


STS tool generates the pom.xml file for us, however I have updated the Spring Framework, AspectJ, SLF4J and Jackson version to the latest one as of today. Most of the part is common and generated automatically, the important point to note is that I have added Jackson JSON libraries in the dependency because we will use that to convert Objects to JSON and vice versa.

web.xml



Servlet-Context.xml (DispatchServlet Context)

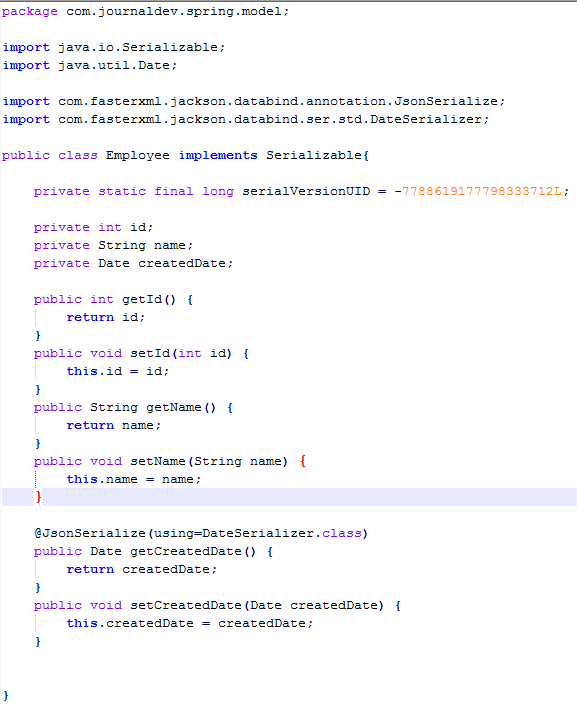


Most of the part is auto generated and contains boiler-plate configurations. However important points to note are **annotation-driven** element to support annotations based configuration and plugging in MappingJackson2HttpMessageConverter to the RequestMappingHandlerAdapter messageConverters so that Jackson API kicks in and converts JSON to Java Beans and vice versa. By having this configuration, we will be using JSON in request body and we will receive JSON data in the response.

Spring REST Model Classes

Employee.java

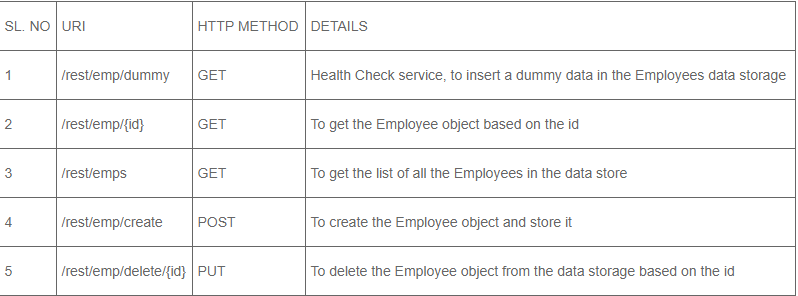
Let’s write a simple POJO class that will serve as input and output to our Restful web service methods.



The only important point to note is the use of @JsonSerialize annotation to use DateSerializer class for Date conversion from Java type to JSON format and vice versa.

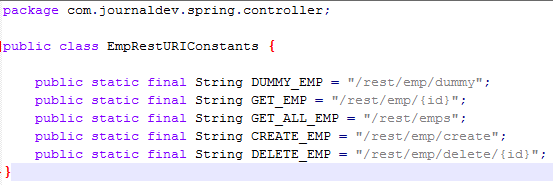
Spring Restful web service End Points

We will have following rest web services end points.



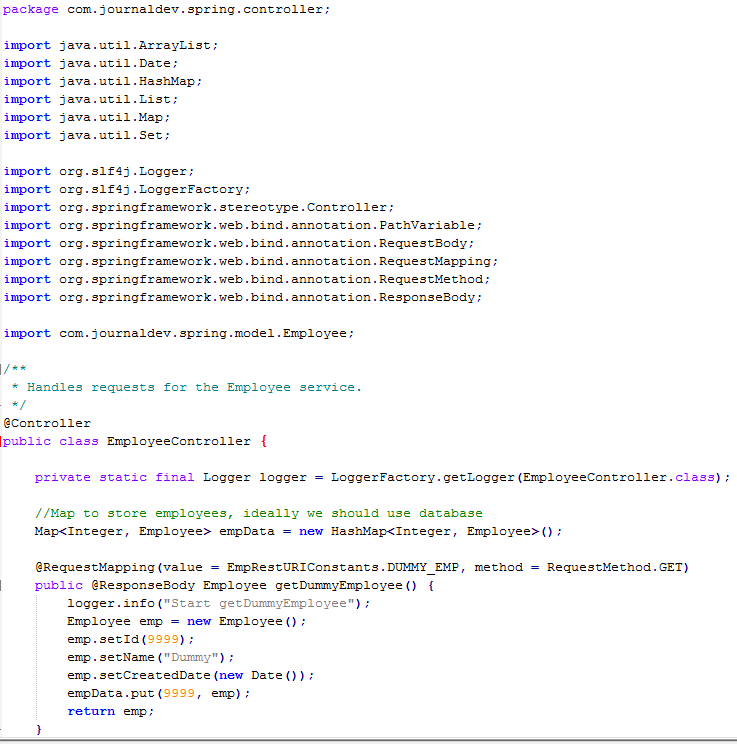
We have a class defining all these URI as String constants.

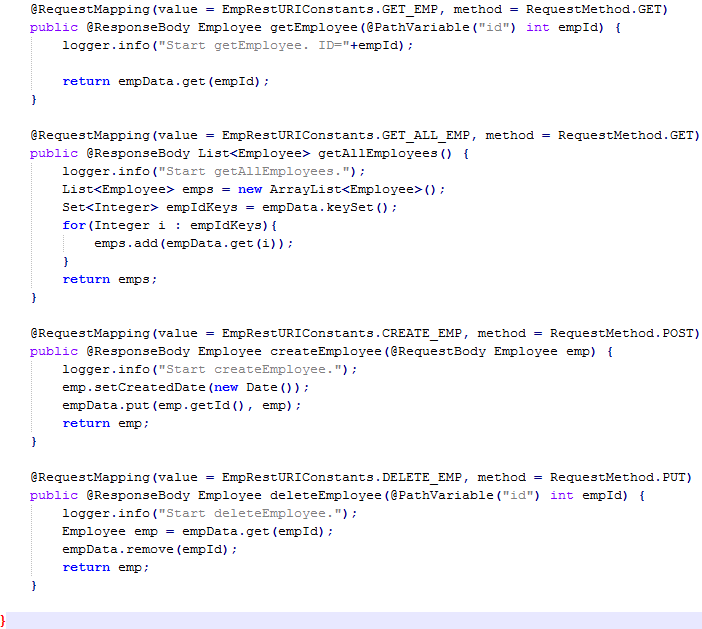
EmpRestURIConstants.java



Spring Restful web service Controller class

Our EmployeeController class will publish all the web service end points mentioned above, let’s look at the code of the class and then we will learn about each of the methods in detail.





For simplicity, I am storing all the employees data in the HashMap empData. @RequestMapping annotation is used to map the request URI to the handler method. We can also specify the HTTP method that should be used by client application to invoke the rest method.

@ResponseBody annotation is used to map the response object in the response body. Once the response object is returned by the handler method, MappingJackson2HttpMessageConverter kicks in and convert it to JSON response.

@PathVariable annotation is the easy way to extract the data from the rest URI and map it to the method argument.

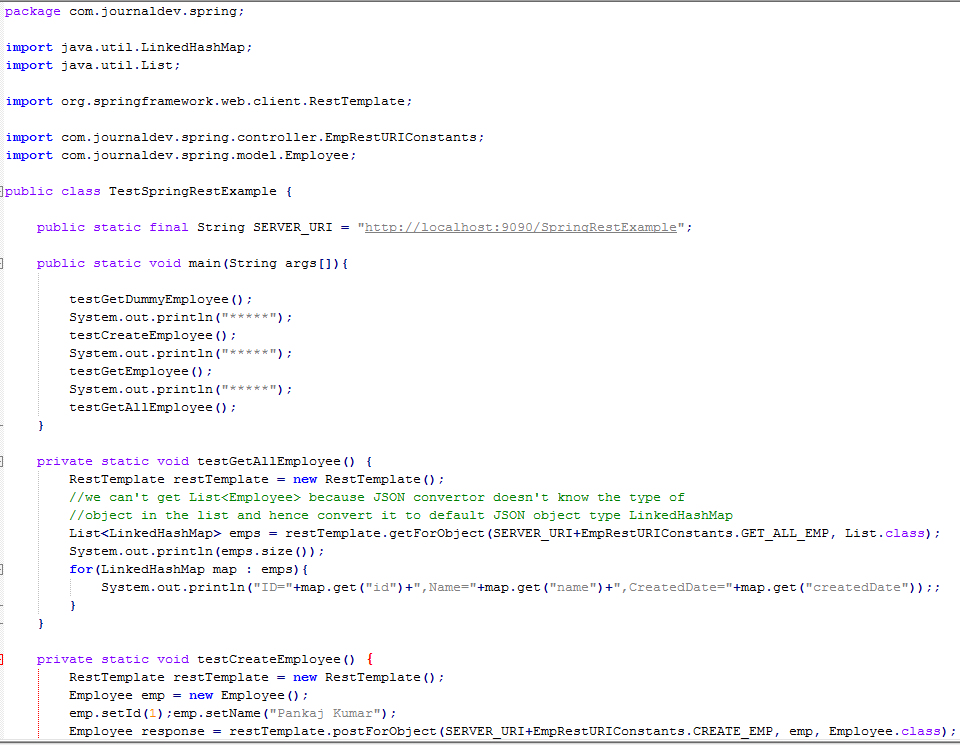
@RequestBody annotation is used to map the request body JSON data into the Employee object, again this is done by the MappingJackson2HttpMessageConverter mapping.

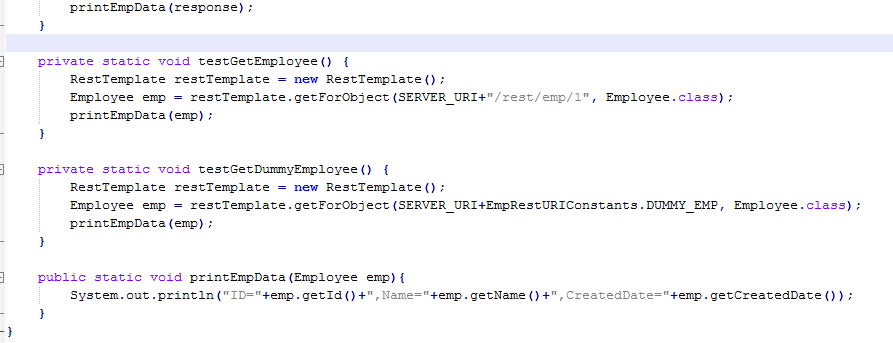
Rest of the code is simple and self understood, our application is ready for deployment and testing. Just export as WAR file and copy it in the servlet container web app directory. If you have server configured in the STS, you can simply run it on the server to get it deployed.

I am using WizTools RestClient to invoke the rest calls but you can also use Chrome extension Postman.

Spring Rest Client Program

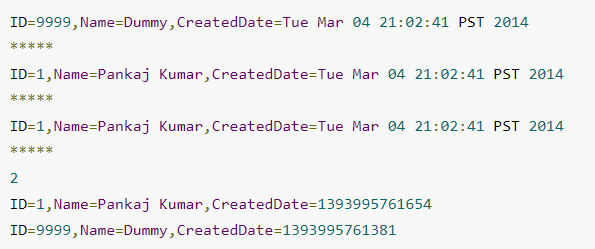
Rest Clients are good to test our rest web service but most of the times, we need to invoke rest services through our program. We can use Spring RestTemplate to invoke these methods easily. Below is a simple program invoking our application rest methods using RestTemplate API.





Most of the program is simple to understand, however when invoking rest method returning a Collection, we need to use LinkedHashMap because JSON to object conversion doesn’t know about the Employee object and converts it to the collection of LinkedHashMap. We can write a utility method to convert from LinkedHashMap to our Java Bean object.

When we run above program, we get following output in the console.



1. Spring MVC form handling tutorial (which same as CYLC situation)

<http://www.codejava.net/frameworks/spring/spring-mvc-form-handling-tutorial-and-example> **This example explain how jsp page interact with Spring controller and model**

(1)Coding Model Class (form-backing object)

Create User.java class with the following code:

File: src/main/java/net/codejava/spring/model/User.java

package net.codejava.spring.model;

import java.util.Date;

public class User {

private String username;

private String password;

private String email;

private Date birthDate;

private String profession;

// getters and setters...

}

As we can see, this model class has five fields (username, password, email, birthDate and profession) which bind to the corresponding fields in the view (JSP page). When an object of a model class is bound to a form, it is called form-backing object.

(2)Coding Registration Form using Spring Form Tags

File: src/main/webapp/WEB-INF/views/Registration.jsp

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

"http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Registration</title>

</head>

<body>

<div align="center">

<form:form action="register" method="post" commandName="userForm">

<table border="0">

<tr>

<td colspan="2" align="center"><h2>Spring MVC Form Demo - Registration</h2></td>

</tr>

<tr>

<td>User Name:</td>

<td><form:input path="username" /></td> 🡪

Note: path specify corresponding property in model class

</tr>

<tr>

<td>Password:</td>

<td><form:password path="password" /></td>

</tr>

<tr>

<td>E-mail:</td>

<td><form:input path="email" /></td>

</tr>

<tr>

<td>Birthday (mm/dd/yyyy):</td>

<td><form:input path="birthDate" /></td>

</tr>

<tr>

<td>Profession:</td>

<td><form:select path="profession" items="${professionList}" /></td>

</tr>

<tr>

<td colspan="2" align="center"><input type="submit" value="Register" /></td>

</tr>

</table>

</form:form>

</div>

</body>

</html>

The <form:form> tag plays an important role here. It’s very similar to the regular HTLM <form> tag but the commandName attribute is the key which specifies name of the model class object that acts as a backing object for this form:

<form:form action="register" method="post" commandName="userForm">

We’ll see how to set a model class object as the form-backing object when coding the controller class.

Also in this JSP page, we are using few Spring form tags to generate equivalent HTML form input tags and bind these form fields with corresponding properties of the model class User above, including:

<form:input>: generates HTML <input type=”text”> tag.

<form:password>: generates HTML <input type=”password”> tag.

<form:select>: generates HTML <select> tag and its items in <option> tags.

Here, the noteworthy attribute of each tag is path - which specifies name of a property of the model class. See a complete list of form tags provided by Spring MVC here.

(3)Coding Controller Class

File: src/main/java/net/codejava/spring/controller/RegisterController.java

package net.codejava.spring.controller;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import net.codejava.spring.model.User;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

@RequestMapping(value = "/register")

public class RegisterController {

@RequestMapping(method = RequestMethod.GET)

public String viewRegistration(Map<String, Object> model) {

User userForm = new User();

model.put("userForm", userForm);

List<String> professionList = new ArrayList<>();

professionList.add("Developer");

professionList.add("Designer");

professionList.add("IT Manager");

model.put("professionList", professionList);

return "Registration";

}

@RequestMapping(method = RequestMethod.POST)

public String processRegistration(@ModelAttribute("userForm") User user,

Map<String, Object> model) {

// implement your own registration logic here...

// for testing purpose:

System.out.println("username: " + user.getUsername());

System.out.println("password: " + user.getPassword());

System.out.println("email: " + user.getEmail());

System.out.println("birth date: " + user.getBirthDate());

System.out.println("profession: " + user.getProfession());

return "RegistrationSuccess";

}

}

We can see that this controller is designed to handle the request URL /register:

@RequestMapping(value = "/register")

We implement two methods viewRegistration() and processRegistration() to handle the GET and POST requests, respectively. Writing handler methods in Spring is very flexible, as we can freely choose our own method names and necessary parameters. Let’s look at each method of the above controller class in details:

viewRegistration(): in this method we create a model object and put it into the model map with the key “userForm”:

User userForm = new User(); 🡪 This create a model object

model.put("userForm", userForm);

This creates a binding between the specified object with the form in the view returned by this method (which is the registration form). Note that the key “userForm” must match value of the commandName attribute of the <form:form> tag.

Another interesting point is that we create a list of Strings and put it into the model map with the key “professionList”:

List<String> professionList = new ArrayList<>();

professionList.add("Developer");

professionList.add("Designer");

professionList.add("IT Manager");

model.put("professionList", professionList);

This collection will be used by the <form:select> tag in the Registration.jsp page in order to render the profession dropdown list dynamically.

Finally this method returns a view name (“Registration”) which will be mapped to the registration form page above.

processRegistration(): this method handles the form submission (via POST request). The important parameter here is:

@ModelAttribute("userForm") User user 🡪 @ModelAttribute on parameter = @Autowired + @Qualified

This will make the model object which is stored under the key “userForm” in the model map available to the method body. Again, the key “userForm” must match value of the commandName attribute of the <form:form> tag.

When the form is submitted, Spring automatically binds the form’s field values to the backing object in the model, thus we can access the form values inputted by the user through this backing object like this

System.out.println("username: " + user.getUsername());

(4)Coding Registration Success Page

File: src/main/webapp/WEB-INF/views/RegistrationSuccess.jsp

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

"http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Registration Success</title>

</head>

<body>

<div align="center">

<table border="0">

<tr>

<td colspan="2" align="center"><h2>Registration Succeeded!</h2></td>

</tr>

<tr>

<td colspan="2" align="center">

<h3>Thank you for registering! Here's the review of your details:</h3>

</td>

</tr>

<tr>

<td>User Name:</td>

<td>${userForm.username}</td>

</tr>

<tr>

<td>E-mail:</td>

<td>${userForm.email}</td>

</tr>

<tr>

<td>Birthday:</td>

<td>${userForm.birthDate}</td>

</tr>

<tr>

<td>Profession:</td>

<td>${userForm.profession}</td>

</tr>

</table>

</div>

</body>

</html>

1. What is difference between model.put() vs model.addAttribute() ?

<http://stackoverflow.com/questions/15742262/modelmap-put-v-s-modelmap-addattribute>

addAttributes implies check for not null in attribute name -> see sources

/\*\*

\* Add the supplied attribute under the supplied name.

\* @param attributeName the name of the model attribute (never <code>null</code>)

\* @param attributeValue the model attribute value (can be <code>null</code>)

\*/

public ModelMap addAttribute(String attributeName, Object attributeValue) {

Assert.notNull(attributeName, "Model attribute name must not be null");

put(attributeName, attributeValue);

return this;

}

1. What is the viewresolver ?

<http://www.journaldev.com/2696/spring-interview-questions-and-answers>

ViewResolver implementations are used to resolve the view pages by name. Usually we configure it in the spring bean configuration file. For example

<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->

<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<beans:property name="prefix" value="/WEB-INF/views/" />

<beans:property name="suffix" value=".jsp" />

</beans:bean>

InternalResourceViewResolver is one of the implementation of ViewResolver interface and we are providing the view pages directory and suffix location through the bean properties. So if a controller handler method returns “home”, view resolver will use view page located at /WEB-INF/views/home.jsp.

1. How is event handling done in Spring ?

<http://howtodoinjava.com/spring/spring-core/top-spring-core-interview-questions-with-answers/>

Spring’s ApplicationContext provides the functionality to support events and listeners in code. We can create beans that listen for events which are published through our ApplicationContext. Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. So if a bean implements the ApplicationListener, then every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified.

public class AllApplicationEventListener implements ApplicationListener < ApplicationEvent >

{

@Override

public void onApplicationEvent(ApplicationEvent applicationEvent)

{

//process event

}

}

We have 5 kind of standard event(almost start/ stop / refresh / close event around context)

ContextRefreshedEvent: This event is published when the ApplicationContext is either initialized or refreshed. This can also be raised using the refresh() method on the ConfigurableApplicationContext interface.

ContextStartedEvent: This event is published when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. You can poll your database or you can re/start any stopped application after receiving this event.

ContextStoppedEvent: This event is published when the ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. You can do required housekeep work after receiving this event.

ContextClosedEvent: This event is published when the ApplicationContext is closed using the close() method on the ConfigurableApplicationContext interface. A closed context reaches its end of life; it cannot be refreshed or restarted.

RequestHandledEvent: This is a web-specific event telling all beans that an HTTP request has been serviced.

And we can create our own customer event by extending ApplicationEvent, create our own customer listener by implementing ApplicationListener

<http://howtodoinjava.com/spring/spring-core/top-spring-core-interview-questions-with-answers/>

1. Define a customer event

public class CustomApplicationEvent extends ApplicationEvent {

public CustomApplicationEvent ( Object source, final String msg )

{

super(source);

System.out.println("Created a Custom event");

}

}

1. Create a listener to listen to this event

public class CustomEventListener implements ApplicationListener < CustomApplicationEvent > {

@Override

public void onApplicationEvent(CustomApplicationEvent applicationEvent) {

//handle event

}

}

1. And to publish this event, you will need the help of applicationContext instance.

CustomApplicationEvent customEvent =

new CustomApplicationEvent( applicationContext, "Test message" );

applicationContext.publishEvent ( customEvent );

1. What is Model in Spring ?

<http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html>

PROBLEM

In the context of Spring MVC, a model generally represents the data that will be passed to and from an operation (defined in a web controller) and the view. Spring MVC supports various options and patterns for defining the data that can be passed back and forth, and a typical Spring MVC application may use a combination of these options. The flexibility of Spring MVC to have fine-grained control for defining models is very powerfull, however along with the flexibility comes considerable variability and complexity in the implemenation using Spring.

SOLUTION(Difference between @RequestParam vs @ModelAttribute)

Through the use of code generation in Skyway Builder, the complexity of Spring models can be abstracted to makes things simpler for the developer.

A controller method in Spring can support zero-to-many input parameters, and the principal mechanisms in Spring for specifying input parameters are the @RequestParam and the @ModelAttribute annotations. The @RequestParam annotation is used to bind individual request parameters, like string and integers, to method parameters in the controller. The @ModelAttribute annotation is used to bind complex objects, like domain objects, data transfer objects and/or form backing objects, to method parameters in a controller.

In Skyway Builder operations are used to define request handlers (controller methods), and operation parameters are simply configured using Input variables. Since these variables are available to be processed by the operation, the developer using Skyway Builder doesn't really need to be concerned with the details of Spring.

Nevertheless here's an overview of what gets generated for input variables added to an operation:

(1)the input variable is emitted as a annotated method parameter

(2)the annotaton is determined by the variable type

(3)primitive variables will be annotated with @RequestParam

(4)complex variables will be annotated with @ModelAttribute

(5)the annotation name is derived from the variable name

A controller method in Spring can also output model data, and the principal mechanism in Spring for specifying output model data is the ModelAndView object. In the event that there isn't any data to be returned by the method, the controller method can simply return a String that represents the view that should be rendered. If the controller method does return data, then a ModalAndView object needs to be instantiated and each output variable is added as a model attribute to the ModelAndView object.

In Skyway Builder you simply need to define output parameters on the Operation, and Skyway Builder will handle all the details regarding how the model data should be emitted.

Here's an overview of what occurs regarding output variables added to an operation:

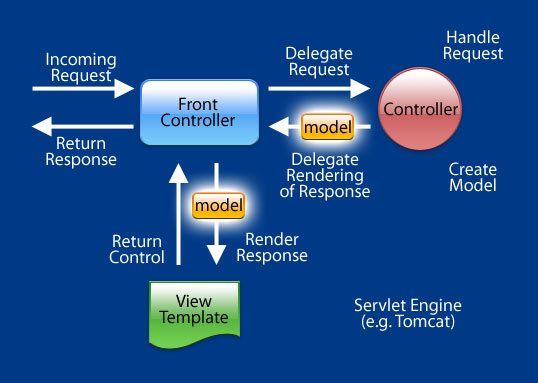
(1)If you don't have any output variables, the method return type will be String, and the last line of the operation will return the corresponding view that should be rendered (as defined in URL mappings).

(2)If you have one or more output variables, the method will return a ModelAndView object, and each output variable will added to the ModelView. The output variable name will be used as the ModelAttribute name, and the ModalAndView will be set with the view that should rendered (as defined in URL mappings).

HOW IT WORKS

Once again a model generally refers to the data that is pased to and from a controller method (Skyway Operation).

Figure 2.5. Model (MVC)



Controller methods parameters are defined using Operation Input and Variables.

Steps for defining operation input and output variables:

From the Skyway Navigator, double-click on the Operation to configure, and switch to the Inputs/Outputs tab.

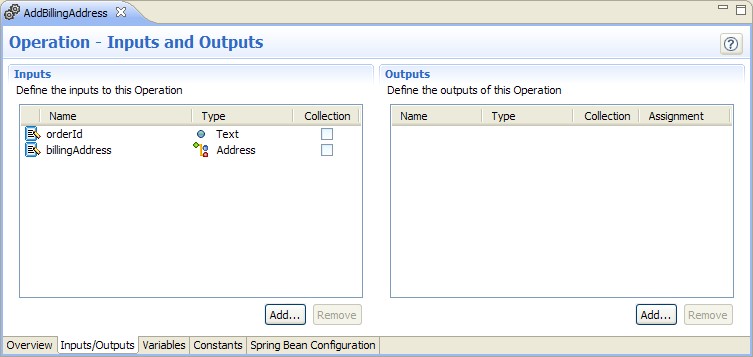
Add input variables by clicking on the Add... button in the Inputs block, and specify the output variable name. Assign (assignment) the variable name to either an input variable or operation variable. The type and collection parameters will be automatically configured based on the selected assignment variable.

Let's examine a few scenarios to see exactly what gets generated based on the configuration of the operation inputs and outputs.

Example 1

In this first example, an operation was defined with two input variables and no output variables.

Figure 2.6. Spring MVC Model (Example 1) - Operation Inputs/Outputs



The following figures shows an abbreviated version of the full controlller method that is generated from the operation configuration.

Example 2.2. Spring MVC Model (Example 1) - Operation Inputs/Outputs (Generated)

@RequestMapping(value = "/OrderCheckoutController/AddBillingAddress.action")

public String[1](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-1.1-out-text) addBillingAddress(@RequestParam("orderId") String orderId[2](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-1.2-out-text),

@ModelAttribute("billingAddress") Address billingAddress[3](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-1.3-out-text)) throws java.lang.Exception {

//method implementation either hand-coded or generated from action model

return "/billingoverview.jsp";[4](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-1.4-out-text)

}

The operation didn't have any defined output variables. Therefore the return type for the controller method is String.



The orderId input variable is of type Text. Since Text is a primitive data type, the method parameter corresponding to this variable is annotated with the @RequestParam annotation. The variable name (orderID) is used as the name.



The billingAddress input variable is of type Address. Since Address is a complex type, the method parameter corresponding to this variable is annotated with the @ModelAttribute annotation. The variable name (billingAddress) is used as the name.



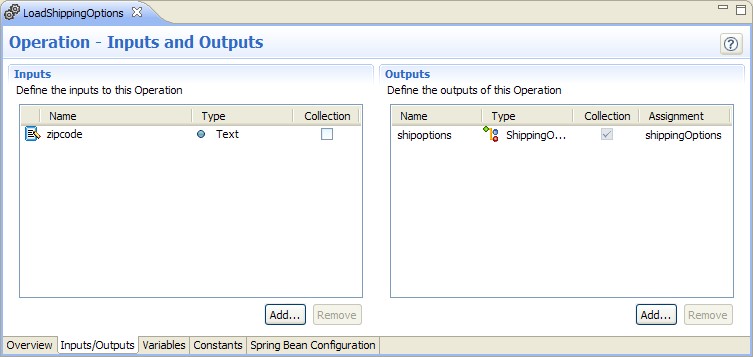
The output of the controller method is set to the path of the view that should be used for rendering the results.



Example 2

In this second example, an operation was defined with one input variable and an output variable. The output variable is mapped to an operation variable (shippingOptions) that is defined in the Variables tab (not shown).

Figure 2.7. Spring MVC Model (Example 2) - Operation Inputs/Outputs



The following figures shows an abbreviated version of the full controlller method that is generated from the operation configuration.

Example 2.3. Spring MVC Model (Example 2) - Operation Inputs/Outputs (Generated)

@RequestMapping(value = "/OrderCheckoutController/LoadShippingOptions.action")

public ModelAndView[1](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-2.1-out-text) loadShippingOptions(@RequestParam("zipcode") String zipcode[2](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-2.2-out-text)) throws java.lang.Exception {

ModelAndView mav = new ModelAndView();[3](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-2.3-out-text)

Set<ShippingOption> shippingOptions = null;

//method implementation either hand-coded or generated from action model

mav.addObject("shipoptions", shippingOptions);[4](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-2.4-out-text)

mav.setViewName("/success.jsp");[5](http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html#model-example-2.5-out-text)

return mav;

}

The operation has atleast one output variable. Therefore the return type for the controller method is ModelAndView.



The zipcode input variable is of type Text. Since Text is a primitive data type, the method parameter corresponding to this variable is annotated with the @RequestParam annotation. The variable name (zipcode) is used as the name.



Since the return type of the controller method is a ModelAndView object, a ModelAndView object is instantiated.



The operation output variables are added to the ModelAndView object. The variable name is used as the model attribute name, and the object (shippingOptions) that was assigned to the output is used as the model.



The view is set on the ModelAndView object to the path of the view that should be used for rendering the results.



1. Data Transfer Objects vs. Form Backing Objects vs. Domain Objects

<http://www.skywayperspectives.org/documentation/6.3/chunk/recipes/ch02s03.html>

In MVC the model is the contract between the View and the Controller. While you can use basic data types, more often than not you'll want to use complex objects for the model.

There are several options available to the developer. You can use your domain objects for the model, but this only works if there's a very close correlation between your domain object and your views. If that's not the case, then you can create a data transfer object that is not tied to your domain model, and the request handlers in the controller would be responsible for mapping data between the data transfer object and the domain model. A data transfer object is a general term for a data object that isn't part of the domain model and isn't intended to be persisted. A form backing object is a specific reference to data object that is used to capture input from an end-user, and it can be implemented using a domain object or a data transfer object.

You can create domain objects and data transfer objects in the Spring DSL using Domain Objects and Component artifacts, respectively. If you prefer, you can also use a custom Java class.

1. Spring Transaction management

<http://www.tutorialspoint.com/spring/spring_transaction_management.htm>

The concept of transactions can be described with following four key properties described as ACID:

Atomicity: A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.

Consistency: This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.

Isolation: There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.

Durability: Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all the four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows:

Begin the transaction using begin transaction command.

Perform various deleted, update or insert operations using SQL queries.

If all the operation are successful then perform commit otherwise rollback all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. The Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs requires an application server, but Spring transaction management can be implemented without a need of application server.

Programmatic transaction management: This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain. <http://www.tutorialspoint.com/spring/programmatic_management.htm>

Let us use PlatformTransactionManager directly to implement programmatic approach to implement transactions. To start a new transaction you need to have a instance of TransactionDefinition with the appropriate transaction attributes. For this example we will simply create an instance of DefaultTransactionDefinition to use the default transaction attributes.

Following is the implementation class file StudentJDBCTemplate.java for the defined DAO interface StudentDAO:

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplateObject;

private PlatformTransactionManager transactionManager;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplateObject = new JdbcTemplate(dataSource);

}

public void setTransactionManager(

PlatformTransactionManager transactionManager) {

this.transactionManager = transactionManager;

}

public void create(String name, Integer age, Integer marks, Integer year){

TransactionDefinition def = new DefaultTransactionDefinition();

TransactionStatus status = transactionManager.getTransaction(def);

try {

String SQL1 = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL1, name, age);

// Get the latest student id to be used in Marks table

String SQL2 = "select max(id) from Student";

int sid = jdbcTemplateObject.queryForInt( SQL2 );

String SQL3 = "insert into Marks(sid, marks, year) " +

"values (?, ?, ?)";

jdbcTemplateObject.update( SQL3, sid, marks, year);

System.out.println("Created Name = " + name + ", Age = " + age);

transactionManager.commit(status);

} catch (DataAccessException e) {

System.out.println("Error in creating record, rolling back");

transactionManager.rollback(status);

throw e;

}

return;

}

public List<StudentMarks> listStudents() {

String SQL = "select \* from Student, Marks where Student.id=Marks.sid";

List <StudentMarks> studentMarks = jdbcTemplateObject.query(SQL,

new StudentMarksMapper());

return studentMarks;

}

}

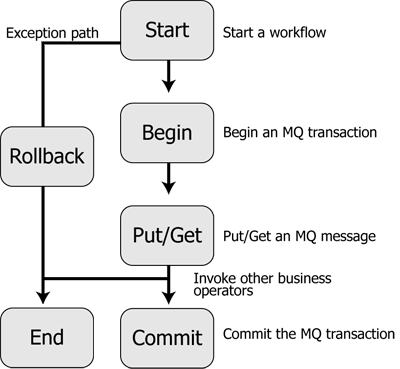
Declarative transaction management: This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

<http://www.tutorialspoint.com/spring/declarative_management.htm>

We use <tx:advice /> tag, which creates a transaction-handling advice and same time we define a pointcut that matches all methods we wish to make transactional and reference the transactional advice.

**A more popular Java Transcation Management interview question**

<http://javadecodedquestions.blogspot.com/2013/03/java-investment-bank-question.html>



**Atomicity**

     Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. Modification on the data in the database either fail or succeed. The beginning of such a modification starts with a transaction and ends when a transaction finishes

**Consistency**

    The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints,cascades, triggers, and any combination thereof.

**Isolation**

    The isolation property ensures that the concurrent execution of transactions results in a system state that could have been obtained if transactions are executed serially, i.e. one after the other. Each transaction has to execute in total isolation i.e. if T1 and T2 are being executed concurrently then both of them should remain unaware of each other's presence.

**Durability**

    Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter).

Transaction Isolation levels

     Transaction isolation levels specify what data is visible to statements within a transaction. These levels directly impact the level of concurrent access by defining what interaction is possible between transactions against the same target data source.

**Database anomalies**

    Database anomalies are generated results that seem incorrect when looked at from the scope of a single transaction, but are correct when looked at from the scope of all transactions. The different types of database anomalies are described as follows:

• **Dirty reads [Read uncommitted]**occur when:

     ◦Transaction A inserts a row into a table.

     ◦Transaction B reads the new row.

     ◦Transaction A rolls back.

     ◦Transaction B may have done work to the system based on the row inserted by transaction A, but that row never became a permanent part of the database.

• **Non-Repeatable reads [Read committed]**occur when:

   ◦ Transaction A reads a row.

   ◦ Transaction B changes the row.

   ◦ Transaction A reads the same row a second time and gets the new results.

• **Phantom reads**occur when:

  ◦ Transaction A reads all rows that satisfy a WHERE clause on an SQL query.

  ◦ Transaction B inserts an additional row that satisfies the WHERE clause.

  ◦ Transaction A re-evaluates the WHERE condition and picks up the additional row.

Transaction isolation level expose the application to the allowable database anomolies at the prescribed levels due to its locking strategies.

JDBC transaction isolation levels

    There are five levels of transaction isolation in the IBM Developer Kit for Java JDBC API. Listed from least to most restrictive, they are as follows:

**TRANSACTION\_NONE**

     This is a special constant indicating that the JDBC driver does not support transactions.

**TRANSACTION\_READ\_UNCOMMITTED**

    This level allows transactions to see uncommitted changes to the data. All database anomalies are possible at this level.

**TRANSACTION\_READ\_COMMITTED**

    This level means that any changes made inside a transaction are not visible outside it until the transaction is committed. This prevents dirty reads from being possible.

**TRANSACTION\_REPEATABLE\_READ**

    This level means that rows that are read retain locks so that another transaction cannot change them when the transaction is not completed. This disallows dirty reads and nonrepeatable reads. Phantom read are still possible.

**TRANSACTION\_SERIALIZABLE**

Tables are locked for the transaction so that WHERE conditions cannot be changed by other transactions that add values to or remove values from a table. This prevents all types of database anomalies.

**Spring Transaction Management:**

In addition to the XML-based declarative approach to transaction configuration, you can also use an annotation-based approach to transaction configuration in Spring. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code, and there is generally not much danger of undue coupling, since code that is meant to be used transactionally is almost always deployed that way anyway.

The ease-of-use afforded by the use of the @Transactional annotation is best illustrated with an example, after which all of the details will be explained. Consider the following

*Class definition:*

*// the service class that we want to make transactional*

***@Transactional***

*public class DefaultFooService implements FooService {*

*Foo getFoo(String fooName);*

*Foo getFoo(String fooName, String barName);*

*void insertFoo(Foo foo);*

*void updateFoo(Foo foo);*

*}*

About @Transactional in Spring background

<http://stackoverflow.com/questions/1099025/spring-transactional-what-happens-in-background>

This is a big topic. The Spring reference doc devotes multiple chapters to it. I recommend reading the ones on Aspect-Oriented Programming and Transactions, as Spring's declarative transaction support uses AOP at its foundation.

When Spring loads your bean definitions, and has been configured to look for @Transactional annotations, it will create these proxy objects around your actual bean. These proxy objects are instances of classes that are auto-generated at runtime. The default behaviour of these proxy objects when a method is invoked is just to invoke the same method on the "target" bean (i.e. your bean).

However, the proxies can also be supplied with interceptors, and when present these interceptors will be invoked by the proxy before it invokes your target bean's method. For target beans annotated with @Transactional, Spring will create a TransactionInterceptor, and pass it to the generated proxy object. So when you call the method from client code, you're calling the method on the proxy object, which first invokes the TransactionInterceptor (which begins a transaction), which in turn invokes the method on your target bean. When the invocation finishes, the TransactionInterceptor commits/rolls back the transaction. It's transparent to the client code.

As for the "external method" thing, if your bean invokes one of its own methods, then it will not be doing so via the proxy. Remember, Spring wraps your bean in the proxy, your bean has no knowledge of it. Only calls from "outside" your bean go through the proxy.

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely one line of XML configuration, like so:

<!-- from the file 'context.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"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

<!-- this is the service object that we want to make transactional -->

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- enable the configuration of transactional behavior based on annotations -->

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

<!-- a PlatformTransactionManager is still required -->

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<!-- (this dependency is defined somewhere else) -->

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

</bean>

<!-- other <bean/> definitions here -->

</beans>

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the DefaultFooService class is annotated at the class level with the settings for a read-only transaction, but the @Transactional annotation on the updateFoo(Foo) method in the same class takes precedence over the transactional settings defined at the class level.

@Transactional(readOnly = true)

public class DefaultFooService implements FooService {

public Foo getFoo(String fooName) {

// do something

}

// these settings have precedence for this method

@Transactional(readOnly = false, propagation = Propagation.REQUIRES\_NEW)

public void updateFoo(Foo foo) {

// do something

}

}

Programmatic Transaction Management – Transaction Manager

This is sample code to manage transaction programmatically.

private PlatformTransactionManager transactionManager;

public void setTransactionManager(PlatformTransactionManager transactionManager){

this.transactionManager = transactionManager;

}

public void performTransaction()

{

TransactionDefinition transactionDefinition = new DefaultTransactionDefinition();

TransactionStatus status = transactionManager.getTransaction(transactionDefinition);

try{

// business processing

transactionManager.commit(status);

}catch (Exception e){

transactionManager.rollback(status);

}

}

1. How to handle exceptions in Spring MVC framework ?

<http://www.tutorialspoint.com/spring/spring_exception_handling_example.htm>

Following is the content of Student.java file:

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of SpringException.java file:

package com.tutorialspoint;

public class SpringException extends RuntimeException{ 🡪 Note: Need one exception class which extends RuntimeException

private String exceptionMsg;

public SpringException(String exceptionMsg) {

this.exceptionMsg = exceptionMsg;

}

public String getExceptionMsg(){

return this.exceptionMsg;

}

public void setExceptionMsg(String exceptionMsg) {

this.exceptionMsg = exceptionMsg;

}

}

Following is the content of StudentController.java file. Here you need to annotate a service method using @ExceptionHandler where you can specify one or more exceptions to be handled. If you are specifying more than one exceptions then you can use comma separated values.

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.ui.ModelMap;

@Controller

public class StudentController {

@RequestMapping(value = "/student", method = RequestMethod.GET)

public ModelAndView student() {

return new ModelAndView("student", "command", new Student());

}

@RequestMapping(value = "/addStudent", method = RequestMethod.POST)

@ExceptionHandler({SpringException.class}) 🡪 Note: ExceptionHandler point to exception class.

public String addStudent( @ModelAttribute("HelloWeb")Student student,

ModelMap model) {

if(student.getName().length() < 5 ){

throw new SpringException("Given name is too short");

}else{

model.addAttribute("name", student.getName());

}

if( student.getAge() < 10 ){

throw new SpringException("Given age is too low");

}else{

model.addAttribute("age", student.getAge());

}

model.addAttribute("id", student.getId());

return "result";

}

}

Following is the content of Spring Web configuration file web.xml

<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">

<display-name>Spring Exception Handling</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file HelloWeb-servlet.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-3.0.xsd

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

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

<bean class="org.springframework.web.servlet.handler.

SimpleMappingExceptionResolver">

Note: SimpleMappingExceptionResolver is a new resolver especially for handling exception than normal usage of InternalResourceViewResolver.

<property name="exceptionMappings"> 🡪 Note: This property will mapping to jsp page

<props>

<prop key="com.tutorialspoint.SpringException">

ExceptionPage

</prop>

</props>

</property>

<property name="defaultErrorView" value="error"/>

</bean>

</beans>

Here you specified ExceptionPage as an exception view in case SpringException occurs, if there is any other type of exception then a generic view error will take place.

Following is the content of Spring view file student.jsp:

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<title>Spring MVC Exception Handling</title>

</head>

<body>

<h2>Student Information</h2>

<form:form method="POST" action="/HelloWeb/addStudent">

<table>

<tr>

<td><form:label path="name">Name</form:label></td>

<td><form:input path="name" /></td>

</tr>

<tr>

<td><form:label path="age">Age</form:label></td>

<td><form:input path="age" /></td>

</tr>

<tr>

<td><form:label path="id">id</form:label></td>

<td><form:input path="id" /></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Submit"/>

</td>

</tr>

</table>

</form:form>

</body>

</html>

Following is the content of Spring view file error.jsp:

<html>

<head>

<title>Spring Error Page</title>

</head>

<body>

<p>An error occured, please contact webmaster.</p>

</body>

</html>;

Following is the content of Spring view file ExceptionPage.jsp. Here you will access the exception instance via ${exception}.

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<title>Spring MVC Exception Handling</title>

</head>

<body>

<h2>Spring MVC Exception Handling</h2>

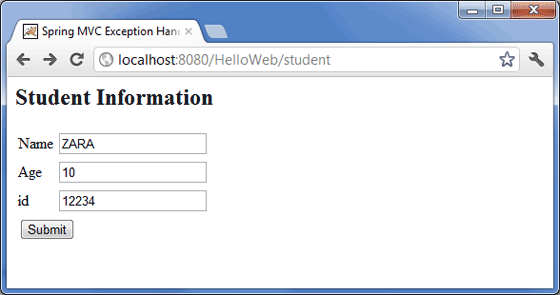
<h3>${exception.exceptionMsg}</h3>

</body>

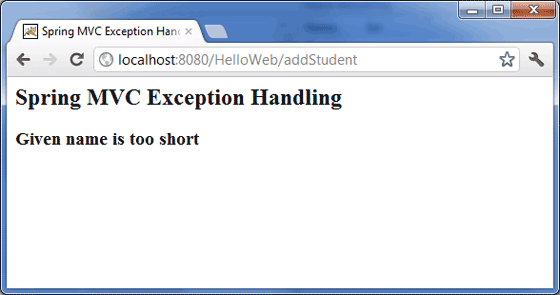
</html>

Once you are done with creating source and configuration files, export your application. Right click on your application and use Export > WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try to access the URL http://localhost:8080/HelloWeb/student. If everything is fine with your Spring Web Applicationand, you should see the following result :



Enter the values as shown above and click submit buttom. If everything is fine with your Spring Web Application, you should see the following result:



1. Spring Transaction Management interview question ?

<http://www.concretepage.com/interview/spring-interview/interview-questions-spring-transaction-management>

|  |
| --- |
| Qns-1: Describe Global and Local transactions in Spring. |
| Ans: Global transactions help to work with multiple transactional resources like relational database and message queue. Global transactions are managed through JTA and JNDI.  Local transactions are resource-specific like JDBC connection. Local Transactions can work with multiple transactional resources. |
|  |
| Qns-2: What is the role of TransactionDefinition interface? |
| Ans: a. Isolation b. Propagation  c. Timeout  d. Read-only status |
|  |
| Qns-3: How can we roll back a declarative transaction? |
| Ans: We can use rollback-for and no-rollback-for attributes with transactional definition. |
|  |
| Qns-4: How many types of isolation are there? |
| Ans: a. ISOLATION\_DEFAULT: default isolation.  b. ISOLATION\_READ\_COMMITTED: dirty reads are prevented, non-repeatable and phantom reads are allowed. c. ISOLATION\_READ\_UNCOMMITTED : dirty reads are allowed, no-repeatable and phantom reads are allowed.  d. ISOLATION\_REPEATABLE\_READ: dirty reads and non-repeatable reads are prevented but phantom reads are allowed.  e. ISOLATION\_SERIALIZABLE : dirty , non- repeatable reads and phantom reads are prevented. |
|  |
| Qns-5: How many types of Propagation are there? |
| Ans: Find the Propagation type. a. PROPAGATION\_MANDATORY : supports current transaction and throws exception if no transaction available.  b. PROPAGATION\_NESTED : runs with nested transaction  c. PROPAGATION\_NEVER : does not run with current transaction and throws exception if current transaction exits.  d. PROPAGATION\_NOT\_SUPPORTED : runs non -transactionaly and does not support current transaction.  e. PROPAGATION\_REQUIRED : runs with current transaction and create one if does not exist.  f. PROPAGATION\_REQUIRES\_NEW : creates new transaction and suspends if exits any.  g. PROPAGATION\_SUPPORTS: runs current transaction and runs non -transactionaly |
|  |

1. What is AOP ?

<http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html>

The core construct of AOP is the aspect, which encapsulates behaviors affecting multiple classes into reusable modules. AOP is a programming technique that allows a developer to modularize crosscutting concerns, that cuts across the typical divisions of responsibility, such as logging and transaction management. Spring AOP, aspects are implemented using regular classes or regular classes annotated with the @Aspect annotation.

1. Explain Advice, joint point and point cut ?

<http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html>

Advice: It’s an implementation of aspect; advice is inserted into an application at join points. Different types of advice include “around,” “before” and “after” advice

Joint Point: On AOP field, similar to Object oriented programming, AOP is another popular programming concept which complements OOPS. A join point is an opportunity within the code for which we can apply an aspect. In Spring AOP, a join point always represents a method execution.

Pointcut: a predicate that matches join points. A point cut is something that defines at what join-points an advice should be applied.

1. How to control concurrent session in Java and Spring ?

<http://javarevisited.blogspot.com/2012/03/spring-security-example-tutorial-how-to.html>

As I said it’s simple and easy when you use spring security framework or library. In fact is all declarative and no code is require to enable **concurrent session disable functionality**. You will need to include following xml snippet in your *Spring Security Configuration file* mostly named as applicaContext-security.xml. Here is sample **spring security Example** of limiting user session in Java web application:

**<session-management** invalid-session-url="/logout.html"**>**  
    **<concurrency-control** max-sessions="1" error-if-maximum-exceeded="true" **/>**  
**</session-management>**  
As you see you can **specify how many concurrent session per user is allowed**, most secure system like online banking portals allow just one authenticate session per user. You can even specify a URL where user will be taken if they submit an invalid session identifier can be used to detect session timeout. Session-management element is used to capture session related stuff. Max-session specify how many concurrent authenticated session is allowed and if error-if-maximum-exceeded set to true it will flag error if user tries to login into another session.

1. How do you set up LDAP authentication active directory in Spring ?

<http://javarevisited.blogspot.com/2011/11/ldap-authentication-active-directory.html>

1. Spring view technology

<http://docs.spring.io/autorepo/docs/spring/3.2.x/spring-framework-reference/html/view.html>

1. Spring with JNDI

<http://www.journaldev.com/2597/spring-datasource-jndi-with-tomcat-example>

1. MicroService with Spring

<https://spring.io/blog/2015/07/14/microservices-with-spring>

<https://www.infoq.com/articles/boot-microservices>

1. How to build up a Spring MVC project(together with Tomcat) ?

<http://crunchify.com/simplest-spring-mvc-hello-world-example-tutorial-spring-model-view-controller-tips/>

### Spring Boot Questions

<https://www.dineshonjava.com/spring-boot-interview-questions-and-answers/>

###### What is Spring Boot?

First of all Spring Boot is not a framework, it is a way to ease to create stand-alone application with minimal or zero configurations. It is approach to develop spring based application with very less configuration. It provides defaults for code and annotation configuration to quick start new spring projects within no time. It leverages existing spring projects as well as Third party projects to develop production ready applications. It provides a set of Starter Pom’s or gradle build files which one can use to add required dependencies and also facilitate auto configuration.

Spring Boot automatically configures required classes depending on the libraries on its classpath. Suppose your application want to interact with DB, if there are Spring Data libraries on class path then it automatically sets up connection to DB along with the Data Source class.

###### What are the advantages of using Spring Boot?

* It is very easy to develop Spring Based applications with Java or Groovy.
* It reduces lots of development time and increases productivity.
* It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
* It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
* It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort
* It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
* It provides CLI (Command Line Interface) tool to develop and test Spring Boot (Java or Groovy) Applications from command prompt very easily and quickly.
* It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
* It provides lots of plugins to work with embedded and in-memory Databases very easily.

###### What are the disadvantages of using Spring Boot?

It is very tough and time consuming process to convert existing or legacy Spring Framework projects into Spring Boot Applications. It is applicable only for brand new/Greenfield Spring Projects.

###### Why is it “opinionated”?

It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort. Due to opinionated view of spring boot, what is required to get started but also we can get out if not suitable for application.  
• Spring Boot uses sensible defaults, “opinions”, mostly based on the classpath contents.  
• For example  
– Sets up a JPA Entity Manager Factory if a JPA implementation is on the classpath.  
– Creates a default Spring MVC setup, if Spring MVC is on the classpath.  
• Everything can be overridden easily  
– But most of the time not needed

###### How does it work? How does it know what to configure?

• Auto-configuration works by analyzing the classpath  
– If you forget a dependency, Spring Boot can’t configure it  
– A dependency management tool is recommended  
– Spring Boot parent and starters make it much easier  
• Spring Boot works with Maven, Gradle, Ant/Ivy  
– Our content here will show Maven

###### How are properties defined? Where?

In spring boot, we have to define properties in the application.properties file exists in classpath of application as follow.  
Example: configure default DataSource bean

database.host=localhost

database.user=admin

###### What is the difference between an embedded container and a WAR?

There is no force to go container less  
– Embedded container is just one feature of Spring Boot  
• Traditional WAR also benefits a lot from Spring Boot  
– Automatic Spring MVC setup, including DispatcherServlet  
– Sensible defaults based on the classpath content  
– Embedded container can be used during development

###### What embedded containers does Spring Boot support?

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers.  
By default the embedded server will listen for HTTP requests on port 8080.

###### What does @EnableAutoConfiguration do? What about @SpringBootApplication?

**@EnableAutoConfiguration annotation** on a Spring Java configuration class  
– Causes Spring Boot to automatically create beans it thinks you need  
– Usually based on classpath contents, can easily override

@Configuration

@EnableAutoConfiguration

public class MyAppConfig {

public static void main(String[] args) {

SpringApplication.run(MyAppConfig.class, args);

}

}

**@SpringBootApplication**was available from Spring Boot 1.2  
It is very common to use @EnableAutoConfiguration, @Configuration, and @ComponentScan together.

@Configuration

@ComponentScan

@EnableAutoConfiguration

public class MyAppConfig {

...

}

**With @SpringBootApplication annotation**

@SpringBootApplication

public class MyAppConfig {

...

}

###### What is a Spring Boot starter POM? Why is it useful?

Starters are a set of convenient dependency descriptors that you can include in your application. The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

The starter POMs are convenient dependency descriptors that can be added to your application’s Maven. In simple words, if you are developing a project that uses Spring Batch for batch processing, you just have to include spring-boot-starter-batch that will import all the required dependencies for the Spring Batch application. This reduces the burden of searching and configuring all the dependencies required for a framework.

###### Spring Boot supports both Java properties and YML files. Would you recognize and understand them if you saw them?

spring boot application java property file name is application.properties  
spring boot application YML file name is application.yml

###### Can you control logging with Spring Boot? How?

Yes, we can control logging with spring boot.  
 **Customizing default Configuration for Logging:**  
By adding logback.xml file to the application we can override the default logging configuration providing by the Spring Boot. This file place in the classpath (src/main/resources) of the application for Spring Boot to pick the custom configuration.

**Spring Boot can control the logging level**  
– Just set it in application.properties  
• Works with most logging frameworks  
– Java Util Logging, Logback, Log4J, Log4J2

logging.level.org.springframework=DEBUG

logging.level.com.acme.your.code=INFO

###### How to reload my changes on Spring Boot without having to restart server?

Include following maven dependency in the application.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

<version>1.2.6.RELEASE</version>

</dependency>

**Automatic restart**  
Applications that use spring-boot-devtools will automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder will be monitored for changes.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

This can be achieved using DEV Tools. With this dependency any changes you save, the embedded tomcat will restart. Spring Boot has a Developer tools (DevTools) module which helps to improve the productivity of developers. One of the key challenge for the Java developers is to auto deploy the file changes to server and auto restart the server. Developers can reload changes on Spring Boot without having to restart my server. This will eliminates the need for manually deploying the changes every time. Spring Boot doesn’t have this feature when it has released it’s first version. This was a most requested features for the developers. The module DevTools does exactly what is needed for the developers. This module will be disabled in the production environment.

###### What is Actuator in Spring Boot?

pring Boot Actuator is a sub-project of Spring Boot. It adds several production grade services to your application with little effort on your part. There are also has many features added to your application out-of-the-box for managing the service in a production (or other) environment. They’re mainly used to expose different types of information about the running application – health, metrics, info, dump, env etc.

###### How to run Spring boot application to custom port?

In application.properties, add following property.

server.port = 8181

###### How to implement security for Spring boot application?

Add spring security starter to the boot application

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

###### What is the configuration file name used by Spring Boot?

The configuration file used in spring boot projects is application.properties. This file is very important where we would over write all the default configurations. Normally we have to keep this file under the resources folder of the project.

###### How to implement Spring web using Spring boot?

**Web Application Convenience**  
• Boot automatically configures  
– A DispatcherServlet & ContextLoaderListener  
– Spring MVC using same defaults as @EnableWebMvc  
• Plus many useful extra features:  
– Static resources served from classpath  
• /static, /public, /resources or /META-INF/resources  
– Templates served from /templates  
• If Velocity, Freemarker, Thymeleaf, or Groovy on classpath  
– Provides default /error mapping  
• Easily overridden  
– Default MessageSource for I18N

###### How to configure database using Spring boot?

 Use either spring-boot-starter-jdbc or spring-boot-starterdata-jpa and include a JDBC driver on classpath  
• Declare properties

spring.datasource.url=jdbc:mysql://localhost/test

spring.datasource.username=dbuser

spring.datasource.password=dbpass

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

– Spring Boot will create a DataSource with properties set  
– Will even use a connection pool if the library is found on the classpath!

Also refer to

<http://www.javainuse.com/spring/SpringBootInterviewQuestions>