**Spring Core -**

**Application Contexts:-**

Spring comes with several flavors of application context. Here are a few that you’ll most likely encounter:

* *AnnotationConfigApplicationContext*—Loads a Spring application context from one or more Java-based configuration classes
* *AnnotationConfigWebApplicationContext*—Loads a Spring web application context from one or more Java-based configuration classes
* *ClassPathXmlApplicationContext*—Loads a context definition from one or more XML files located in the classpath, treating context-definition files as classpath resources
* *FileSystemXmlApplicationContext*—Loads a context definition from one or more XML files in the filesystem
* *XmlWebApplicationContext*—Loads context definitions from one or more XML files contained in a web application

eg.

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

If you’d rather load your application context from a Java configuration, you can use A*nnotationConfigApplicationContext*:

ApplicationContext context = new AnnotationConfigApplicationContext(

com.springinaction.knights.config.KnightConfig.class);

**Bean Wiring: -**

***Spring Configuration Options:-***

When it comes to expressing a bean wiring specification, Spring is incredibly flexible, offering three primary wiring mechanisms:

* Explicit configuration in XML
* Explicit configuration in Java
* Implicit bean discovery and automatic wiring
  + *Component scanning*—Spring automatically discovers beans to be created in the application context.
  + *Autowiring*—Spring automatically satisfies bean dependencies.

***@Component:-***

This simple annotation identifies this class as a component class and serves as a clue to Spring that a bean should be created for the class.

Component scanning isn’t turned on by default, however. You’ll still need to write an explicit configuration to tell Spring to seek out classes annotated with @*Component*and to create beans from them.

package soundsystem;

import org.springframework.stereotype.Component;

@Component

public class SgtPeppers implements CompactDisc {}

***@ComponentScan***- enable component scanning in Spring.

All beans in a Spring application context are given an ID and one derived from its class name. If you’d rather give the bean a different ID, all you have to do is pass the desired ID as a value to the @*Component* annotation.

@Component("lonelyHeartsClub")

public class SgtPeppers implements CompactDisc {

...

}

***@Configuration:***

The key to creating a JavaConfig class is to annotate it with @*Configuration*. The @*Configuration* annotation identifies this as a configuration class, and it’s expected to contain details on beans that are to be created in the Spring application context.

import org.springframework.context.annotation.Configuration;

@Configuration

public class CDPlayerConfig {

}

***@ComponentScan or <context:component-scan> :-***

@*ComponentScan*- enable component scanning in Spring.

@*ComponentScan* will default to scanning the **same package** and any subpackages underneath it as the configuration class.

package soundsystem;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan

public class CDPlayerConfig { ... }

To specify a **different base package**, you only need to specify the package in @ComponentScan’s value attribute:

@Configuration

@ComponentScan("soundsystem")

public class CDPlayerConfig {}

or

@Configuration

@ComponentScan(basePackages="soundsystem")

public class CDPlayerConfig {}

For **multiple base packages**, set basePackages to an array of packages to be scanned:

@Configuration

@ComponentScan(basePackages={"soundsystem", "com.videos"})

public class CDPlayerConfig {}

@*ComponentScan* also offers you the option of specifying them **via classes or interfaces** that are in the packages:

@Configuration

@ComponentScan(basePackageClasses={CDPlayer.class, DVDPlayer.class})

public class CDPlayerConfig {}

If you’d rather turn on component scanning **via XML configuration**, then you can use the *<context:component-scan>* element from Spring’s context namespace

<?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.xsd

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

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

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

</beans>

***@ContextConfiguration***

The @ContextConfiguration annotation tells it to load its configuration from aJava class.

import static org.junit.Assert.\*;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes=CDPlayerConfig.class)

public class CDPlayerTest {

@Autowired

private CompactDisc cd;

@Test

public void cdShouldNotBeNull() {

assertNotNull(cd);

}

}

The *@ContextConfiguration* annotation tells it to load its configuration from the CDPlayerConfig class. Because that configuration class includes @ComponentScan, the resulting application context should include the CompactDisc bean.

***@Autowired***

Autowiring is a means of letting Spring automatically satisfy a bean’s dependencies by finding other beans in the application context that are a match to the bean’s needs.

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

import org.springframework.stereotype.Component;

@Component

public class CDPlayer implements MediaPlayer {

private CompactDisc cd;

@Autowired

public CDPlayer(CompactDisc cd) {

this.cd = cd;

}

. . .

}

The @*Autowired* annotation’s use isn’t limited to constructors. It can also be used on a property’s **setter method**.

@Autowired

public void setCompactDisc(CompactDisc cd) {

this.cd = cd;

}

Actually, there’s nothing special about setter methods. @Autowired can also be applied on **any method** on the class.

@Autowired

public void insertDisc(CompactDisc cd) {

this.cd = cd;

}

If there are **no matching beans**, Spring will throw an exception as the application context is being created. To **avoid that exception**, you can set the required attribute on @**Autowired to false**:

@Autowired(required=false)

public CDPlayer(CompactDisc cd) {

this.cd = cd;

}

**Note**:- Leaving the property unwired could lead to NullPointer-Exceptions if you don’t check for null in your code.

***Wiring Bean with Java:-***

JavaConfig is the preferred option for explicit configuration because it’s more powerful, type-safe, and refactor-friendly.

**Creating a configuration class:**

The key to creating a JavaConfig class is to annotate it with @Configuration. The @Configuration annotation identifies this as a configuration class, and it’s expected to contain details on beans that are to be created in the Spring application context.

import org.springframework.context.annotation.Configuration;

@Configuration

public class CDPlayerConfig { ... }

**Declaring a simple bean:**

To declare a bean in JavaConfig, you write a method that creates an instance of the desired type and annotate it with **@Bean**.

@Bean

public CompactDisc sgtPeppers() {

return new SgtPeppers();

}

Note : - Here sgtPeppers method name could be bean ID for SgtPeppers class.

If you’d rather it have a different name, you can either rename the method or prescribe a different name with the *name*attribute:

@Bean(name="lonelyHeartsClubBand")

public CompactDisc sgtPeppers() {

return new SgtPeppers();

}

**Injecting with JavaConfig:**

The simplest way to wire up beans in JavaConfig is to refer to the referenced bean’s method with **@Bean** annotation.

In CDPlayerConfig Class

@Bean

public CDPlayer cdPlayer() {

return new CDPlayer(sgtPeppers());

}

**As perviously**

@Bean

public CompactDisc sgtPeppers() {

return new SgtPeppers();

}

**&&**

public class SgtPeppers implements CompactDisc {}

**&&**

public class CDPlayer implements MediaPlayer {

private CompactDisc cd;

@Autowired

public CDPlayer(CompactDisc cd) {

this.cd = cd;

}

. . .

}

Similarly, multiple instances can be created by using different method name which act as bean ID.

@Bean

public CDPlayer cdPlayer() {

return new CDPlayer(sgtPeppers());

}

@Bean

public CDPlayer anotherCDPlayer() {

return new CDPlayer(sgtPeppers());

}

***Referencing XML configuration in JavaConfig:-***

Referencing **another java config file from one config file**.

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Import;

@Configuration

@Import(CDConfig.class)

public class CDPlayerConfig {

@Bean

public CDPlayer cdPlayer(CompactDisc compactDisc) {

return new CDPlayer(compactDisc);

}

}

Referencing **multiple java config file from one config file**.

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Import;

@Configuration

@Import({CDPlayerConfig.class, CDConfig.class})

public class SoundSystemConfig {. . .}

Referencing **xml config file from a java config file**.

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Import;

import org.springframework.context.annotation.ImportResource;

@Configuration

@Import(CDPlayerConfig.class)

@ImportResource("classpath:cd-config.xml")

public class SoundSystemConfig { . . . }

***Referencing JavaConfig in XML configuration:-***

<?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:c="http://www.springframework.org/schema/c"

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

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

**For java config file:**

<bean class="soundsystem.CDConfig" />

**For xml config file:**

<import resource="cdplayer-config.xml" />

</beans>

**Advance Wiring:-**

***Configuring profile beans:-***

**Configuring profiles in Java –**

Spring’s solution for environment-specific beans. In Java configuration, you can use the ***@Profile*** annotation to specify which profile a bean belongs to.

*@Profile* annotation in class level.

package com.myapp;

import javax.activation.DataSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration

@Profile("dev")

public class DevelopmentProfileConfig {

@Bean(destroyMethod="shutdown")

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript("classpath:schema.sql")

.addScript("classpath:test-data.sql")

.build();

}

}

Meanwhile, you may have another configuration class for production that looks like this:

package com.myapp;

import javax.activation.DataSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration

@Profile("prod")

public class ProductionProfileConfig {

@Bean

public DataSource dataSource() {

JndiObjectFactoryBean jndiObjectFactoryBean =

new JndiObjectFactoryBean();

jndiObjectFactoryBean.setJndiName("jdbc/myDS");

jndiObjectFactoryBean.setResourceRef(true);

jndiObjectFactoryBean.setProxyInterface(

javax.sql.DataSource.class);

return (DataSource) jndiObjectFactoryBean.getObject();

}

}

@Profile annotation on method level.

package com.myapp;

import javax.activation.DataSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration

public class DataSourceConfig {

@Bean(destroyMethod="shutdown")

@Profile("dev")

public DataSource embeddedDataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript("classpath:schema.sql")

.addScript("classpath:test-data.sql")

.build();

}

@Bean

@Profile("prod")

public DataSource jndiDataSource() {

JndiObjectFactoryBean jndiObjectFactoryBean =

new JndiObjectFactoryBean();

jndiObjectFactoryBean.setJndiName("jdbc/myDS");

jndiObjectFactoryBean.setResourceRef(true);

jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);

return (DataSource) jndiObjectFactoryBean.getObject();

}

}

**Configuring profiles in 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:jdbc="http://www.springframework.org/schema/jdbc"

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

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

xsi:schemaLocation="

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

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

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

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

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

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

<beans profile="dev">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:schema.sql" />

<jdbc:script location="classpath:test-data.sql" />

</jdbc:embedded-database>

</beans>

<beans profile="qa">

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:url="jdbc:h2:tcp://dbserver/~/test"

p:driverClassName="org.h2.Driver"

p:username="sa"

p:password="password"

p:initialSize="20"

p:maxActive="30" />

</beans>

<beans profile="prod">

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/myDatabase"

resource-ref="true" proxy-interface="javax.sql.DataSource" />

</beans>

</beans>

**Activating profiles –**

Spring honors two separate properties when determining which profiles are active: spring.profiles.active and spring.profiles.default.

There are several ways to set these properties:

* As initialization parameters on DispatcherServlet
* As context parameters of a web application
* As JNDI entries
* As environment variables
* As JVM system properties
* Using the @ActiveProfiles annotation on an integration test class

Setting default properties in web application’s web.xml file.

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

<web-app version="2.5"

xmlns="http://java.sun.com/xml/ns/javaee"

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

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

<context-param>

<param-name>spring.profiles.default</param-name>

<param-value>dev</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>spring.profiles.default</param-name>

<param-value>dev</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Spring offers the @ActiveProfiles annotation to let you specify which profile(s) should be active when a test is run.

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes={PersistenceTestConfig.class})

@ActiveProfiles("dev")

public class PersistenceTest {

...

}

***Conditional Beans:-***

Suppose you want one or more beans to be configured if and only if some library is available in the application’s classpath. Or let’s say you want a bean to be created only if a certain other bean is also declared. Maybe you want a bean to be created if and only if a specific environment variable is set.

Spring 4 introduced a new @Conditional annotation that can be applied to @Bean methods. If the prescribed condition evaluates to true, then the bean is created. Otherwise the bean is ignored.

@Bean

@Conditional(MagicExistsCondition.class)

public MagicBean magicBean() {

return new MagicBean();

}

Here MagicExistsCondition is a class that implements ***Condition*** interface.

package com.habuma.restfun;

import org.springframework.context.annotation.Condition;

import org.springframework.context.annotation.ConditionContext;

import org.springframework.core.type.AnnotatedTypeMetadata;

import org.springframework.util.ClassUtils;

public class MagicExistsCondition implements Condition {

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata){

Environment env = context.getEnvironment();

return env.containsProperty("magic");

}

}

***Condition Interface:-***

public interface Condition {

boolean matches(ConditionContext ctxt, AnnotatedTypeMetadata metadata);

}

*ConditionContext*is an interface that looks something like this:

public interface ConditionContext {

BeanDefinitionRegistry getRegistry();

ConfigurableListableBeanFactory getBeanFactory();

Environment getEnvironment();

ResourceLoader getResourceLoader();

ClassLoader getClassLoader();

}

From the *ConditionContext*, you can do the following:

* Check for bean definitions via the BeanDefinitionRegistry returned from getRegistry().
* Check for the presence of beans, and even dig into bean properties via the ConfigurableListableBeanFactory returned from getBeanFactory().
* Check for the presence and values of environment variables via the Environment retrieved from getEnvironment().
* Read and inspect the contents of resources loaded via the ResourceLoader returned from getResourceLoader().
* Load and check for the presence of classes via the ClassLoader returned from getClassLoader().

As for the AnnotatedTypeMetadata, it offers you a chance to inspect annotations that may also be placed on the @Bean method. Like ConditionContext, Annotated- TypeMetadata is an interface. It looks like this:

public interface AnnotatedTypeMetadata {

boolean isAnnotated(String annotationType);

Map<String, Object>getAnnotationAttributes(String annotationType);

Map<String, Object>getAnnotationAttributes(

String annotationType, boolean classValuesAsString);

MultiValueMap<String, Object>getAllAnnotationAttributes(

String annotationType);

MultiValueMap<String, Object>getAllAnnotationAttributes(

String annotationType, boolean classValuesAsString);

}

Using the isAnnotated() method, you can check to see if the @Bean method is annotated with any particular annotation type. Using the other methods, you can check on the attributes of any annotation applied to the @Bean method.

**@Profile** is itself annotated with **@Conditional** and refers to Profile-Condition as the Condition implementation.

***Addressing ambiguity in Autowiring:-***

**Problem:-**

To illustrate autowiring ambiguity, suppose you’ve annotated the following set-Dessert() method with @Autowired:

@Autowired

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

In this example, Dessert is an interface and is implemented by three classes: Cake, Cookies, and IceCream:

@Component

public class Cake implements Dessert { ... }

@Component

public class Cookies implements Dessert { ... }

@Component

public class IceCream implements Dessert { ... }

Because all three implementations are annotated by @Component, they’re all picked up during component-scanning and created as beans in the Spring application context. Then, when Spring tries to autowire the Dessert parameter in setDessert(), it doesn’t have a single, unambiguous choice. Although most people wouldn’t have any problem making choices when faced with multiple dessert options, Spring can’t choose. Spring has no option but to fail and throw an exception. To be precise, Spring throws a NoUniqueBeanDefinitionException:

**Solution:-**

**1. Designating a primary bean:**

**@Primary**can be used either alongside @Component for beans that are component-scanned or alongside @Bean for beans declared in Java configuration. For example, here’s how you might declare the @Component-annotated IceCream bean as the primary choice:

@Component

@Primary

public class IceCream implements Dessert { ... }

Or, if you’re declaring the IceCream bean explicitly in Java configuration, the @Bean method might look like this:

@Bean

@Primary

public Dessert iceCream() {

return new IceCream();

}

If you’re configuring your beans in XML, you’re not left out. The <bean> element has a primary attribute to specify a primary bean:

<bean id="iceCream"class="com.desserteater.IceCream"primary="true" />

**2.Qualifying autowired beans:**

The limitation of primary beans is that @Primary doesn’t limit the choices to a single unambiguous option. It only designates a preferred option. The @Qualifier annotation is the main way to work with qualifiers. It can be applied alongside @Autowired or @Inject at the point of injection to specify which bean you want to be injected.

@Autowired

@Qualifier("iceCream")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

@Qualifier("iceCream") refers to the bean that has the String “iceCream” as a qualifier.

**Creating custom qualifiers:-**

Instead of relying on the bean ID as the qualifier, you can assign your own qualifier to a bean.

@Component

@Qualifier("cold")

public class IceCream implements Dessert { ... }

or for @Bean annotation in method

@Bean

@Qualifier("cold")

public Dessert iceCream() {

return new IceCream();

}

Can be autowired as

@Autowired

@Qualifier("cold")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

**Defining custom qualifier annotations:**

Perhaps the Popsicle class could also use another @Qualifier:

@Component

@Qualifier("cold")

@Qualifier("fruity")

public class Popsicle implements Dessert { ... }

And at the injection point, you could narrow it down to IceCream like this:

@Autowired

@Qualifier("cold")

@Qualifier("creamy")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

**There’s only one small problem:**Java doesn’t allow multiple annotations of the same type to be repeated on the same item.

What you can do, however, is create custom qualifier annotations to represent the traits you want your beans to be qualified with. Rather than use @Qualifier("cold"), you can use a custom @Cold annotation that’s defined like this:

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,

ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Cold { }

Likewise, you can create a new @Creamy annotation as a replacement for @Qualifier("creamy"):

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,

ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Creamy { }

Now you can revisit IceCream and annotate it with @Cold and @Creamy, like this:

@Component

@Cold

@Creamy

public class IceCream implements Dessert { ... }

**Scoping Beans – Beans Scope**

By default, all beans created in the Spring application context are created as singletons. That is to say, no matter how many times a given bean is injected into other beans, it’s always the same instance that is injected each time.

But sometimes you may find yourself working with a mutable class that does maintain some state and therefore isn’t safe for reuse.

Spring defines several scopes under which a bean can be created, including the following:

* *Singleton*—One instance of the bean is created for the entire application.
* *Prototype*—One instance of the bean is created every time the bean is injected into or retrieved from the Spring application context.
* *Session*—In a web application, one instance of the bean is created for each session.
* *Request*—In a web application, one instance of the bean is created for each request.

If you’re relying on component-scanning to discover and declare a bean, then you can annotate the bean class with @Scope to make it a prototype bean:

@Component

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public class Notepad { ... }

Alternatively, if you’re configuring the Notepad bean as a prototype in Java configuration, you can use @Scope along with @Bean to specify the desired scoping:

@Bean

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public Notepad notepad() {

return new Notepad();

}

And, in the event that you’re configuring the bean in XML, you can set the scope using the scope attribute of the <bean> element:

<bean id="notepad" class="com.myapp.Notepad" scope="prototype" />

***Working with request and session scope:***

@Component

@Scope(

value=WebApplicationContext.SCOPE\_SESSION,

proxyMode=ScopedProxyMode.INTERFACES)

public ShoppingCart cart() { ... }

Here you’re setting the value attribute to the SCOPE\_SESSION constant from Web-ApplicationContext (which has a value of session). This tells Spring to create an instance of the ShoppingCart bean for each ession in a web application.

Notice that @Scope also has a proxyMode attribute set to ScopedProxyMode.INTERFACES.

Suppose you want to inject the ShoppingCart bean into the following setter method on a singletonStoreService bean:

@Component

public class StoreService {

@Autowired

public void setShoppingCart(ShoppingCart shoppingCart) {

this.shoppingCart = shoppingCart;

}

...

}

Because StoreService is a singleton bean, it will be created as the Spring application context is loaded. As it’s created, Spring will attempt to inject ShoppingCart into the setShoppingCart() method. But the ShoppingCart bean, being session scoped, doesn’t exist yet. There won’t be an instance of ShoppingCart until a user comes along and a session is created.

Instead of injecting the actual ShoppingCart bean into StoreService, Spring should inject a proxy to the ShoppingCart bean. This proxy will expose the same methods as ShoppingCart so that for all StoreService knows, it *is* the shopping cart. But when StoreService calls methods on ShoppingCart, the proxy will lazily resolve it and delegate the call to the actual session-scoped Shopping- Cart bean.

Now let’s take this understanding of scoped proxies and discuss the proxyMode attribute. As configured, proxyMode is set to ScopedProxyMode.INTERFACES, indicating that the proxy should implement the ShoppingCart interface and delegate to the implementation bean.

This is fine (and the most ideal proxy mode) as long as ShoppingCart is an interface and not a class. But if ShoppingCart is a concrete class, there’s no way Spring can create an interface-based proxy. So, if the bean type is a concrete class, you must set proxyMode to ScopedProxy- Mode.TARGET\_CLASS to indicate that the proxy should be generated as an extension of the target class.

Same is the case for **request scope**.

***Declaring scoped proxies in XML:***

To set the proxy mode, you must use a new element from Spring’s aop namespace:

<bean id="cart" class="com.myapp.ShoppingCart" scope="session">

<aop:scoped-proxy />

</bean>

<aop:scoped-proxy> is the Spring XML configuration’s counterpart to the @Scopeannotation’s proxyMode attribute. By default, it uses CGLib to create a target class proxy. But you can ask it to generate an interface-based proxy by setting the proxy-target-class attribute to false:

<bean id="cart" class="com.myapp.ShoppingCart" scope="session">

<aop:scoped-proxy proxy-target-class="false" />

</bean>

In order to use the <aop:scoped-proxy> element, you must declare Spring’s aop namespace in your XML configuration:

***Injecting External Values:-***

**Reading from a property files:**

package com.soundsystem;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

@Configuration

@PropertySource("classpath:/com/soundsystem/app.properties")

public class ExpressiveConfig {

@Autowired

Environment env;

@Bean

public BlankDisc disc() {

return new BlankDisc(env.getProperty("disc.title"), env.getProperty("disc.artist"));

}

}

In this example, @PropertySource references a file named app.properties in the classpath. It might look something like this:

disc.title=Sgt. Peppers Lonely Hearts Club Band

disc.artist=The Beatles

getProperty() is overloaded into four variations:

* String getProperty(String key)
* String getProperty(String key, String defaultValue)
* T getProperty(String key, Class<T> type)
* T getProperty(String key, Class<T> type, T defaultValue)

Eg.

int connectionCount = env.getProperty("db.connection.count", Integer.class, 30);

If you use either of the getProperty() methods without specifying a default value, you’ll receive null if the property isn’t defined.

If you want to require that the property be defined, you can use getRequiredProperty() like this:

@Bean

public BlankDisc disc() {

return new BlankDisc(

env.getRequiredProperty("disc.title"),

env.getRequiredProperty("disc.artist"));

}

Here, if either the disc.title property or the disc.artist property is undefined, an IllegalStateException will be thrown.

If you want to check for the existence of a property, you can call containsProperty() on Environment:

boolean titleExists = env.containsProperty("disc.title");

**Resolving Property Placeholders**

In Spring wiring, placeholder values are property names wrapped with ${ ... }.

<bean id="sgtPeppers" class="soundsystem.BlankDisc"

c:\_title="${disc.title}"

c:\_artist="${disc.artist}" />

You can use the @Value annotation in much the same way as you might use the @Autowired annotation. In the BlankDisc class, for example, the constructor might be written like this:

public BlankDisc( @Value("${disc.title}") String title,

@Value("${disc.artist}") String artist) {

this.title = title;

this.artist = artist;

}

In order to use placeholder values, you must configure either a PropertyPlaceholderConfigurer bean or a PropertySourcesPlaceholderConfigurer bean. Starting with Spring 3.1, PropertySourcesPlaceholderConfigurer is preferred because it resolves placeholders against the Spring Environment and its set of property sources. The following @Bean method configures PropertySourcesPlaceholderConfigurer in Java configuration:

@Bean

Public static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {

return new PropertySourcesPlaceholderConfigurer();

}

If you’d rather use XML configuration, the <context:property-placeholder> element from Spring’s context namespace will give you a PropertySourcesPlaceholderConfigurer bean:

<?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.xsd

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

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

<context:property-placeholder />

</beans>

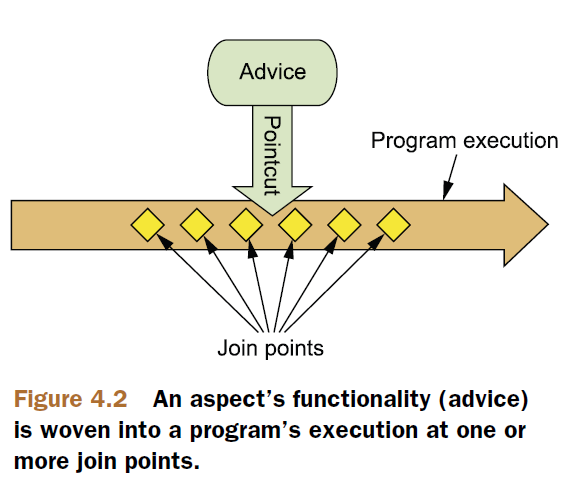
**ASPECT-ORIENTED Spring**

**What is aspect-oriented programming?**

As stated earlier, aspects help to modularize cross-cutting concerns. In short, a crosscutting concern can be described as any functionality that affects multiple points of an application.

With AOP, you still define the common functionality in one place, but you can declaratively define how and where this functionality is applied without having to modify the class to which you’re applying the new feature. Crosscutting concerns can now be modularized into special classes called *aspects*. This has two benefits. First, the logic for each concern is in one place, as opposed to being scattered all over the code base. Second, your service modules are cleaner because they only contain code for their primary concern (or core functionality), and secondary concerns have been moved to aspects.

**Defining AOP terminology:**



Aspects are often described in terms of advice, pointcuts, and join points.

***Advice:***

Aspects have a purpose—a job they’re meant to do. In AOP terms, the job of an aspect is called *advice*. Advice defines both the whatand the *when* of an aspect.

Spring aspects can work with five kinds of advice:

* *Before*—The advice functionality takes place before the advised method is invoked.
* *After*—The advice functionality takes place after the advised method completes, regardless of the outcome.
* *After-returning*—The advice functionality takes place after the advised method successfully completes.
* *After-throwing*—The advice functionality takes place after the advised method throws an exception.
* *Around*—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

***Join Points:***

A *join point* is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modified. These are the points where your aspect’s code can be inserted into the normal flow of your application to add new behavior.

***Pointcuts:***

An aspect doesn’t necessarily advise all join points in an application. *Pointcuts* help narrow down the join points advised by an aspect. If advice defines the *what* and *when* of aspects, then pointcuts define the *where*. A pointcut definition matches one or more join points at which advice should be woven.

***Aspects:***

An *aspect* is the merger of advice and pointcuts. Taken together, advice and pointcuts define everything there is to know about an aspect—what it does and where and when it does it.

***Introductions:***

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

***Weaving:***

*Weaving* is the process of applying aspects to a target object to create a new proxied object. The aspects are woven into the target object at the specified join points. The weaving can take place at several points in the target object’s lifetime:

* *Compile time*—Aspects are woven in when the target class is compiled. This requires a special compiler. AspectJ’s weaving compiler weaves aspects this way.
* *Class load time*—Aspects are woven in when the target class is loaded into the JVM. This requires a special ClassLoader that enhances the target class’s bytecode before the class is introduced into the application. AspectJ 5’s *load-time weaving* (LTW) support weaves aspects this way.
* *Runtime*—Aspects are woven in sometime during the execution of the application. Typically, an AOP container dynamically generates a proxy object that delegates to the target object while weaving in the aspects. This is how Spring AOP aspects are woven.

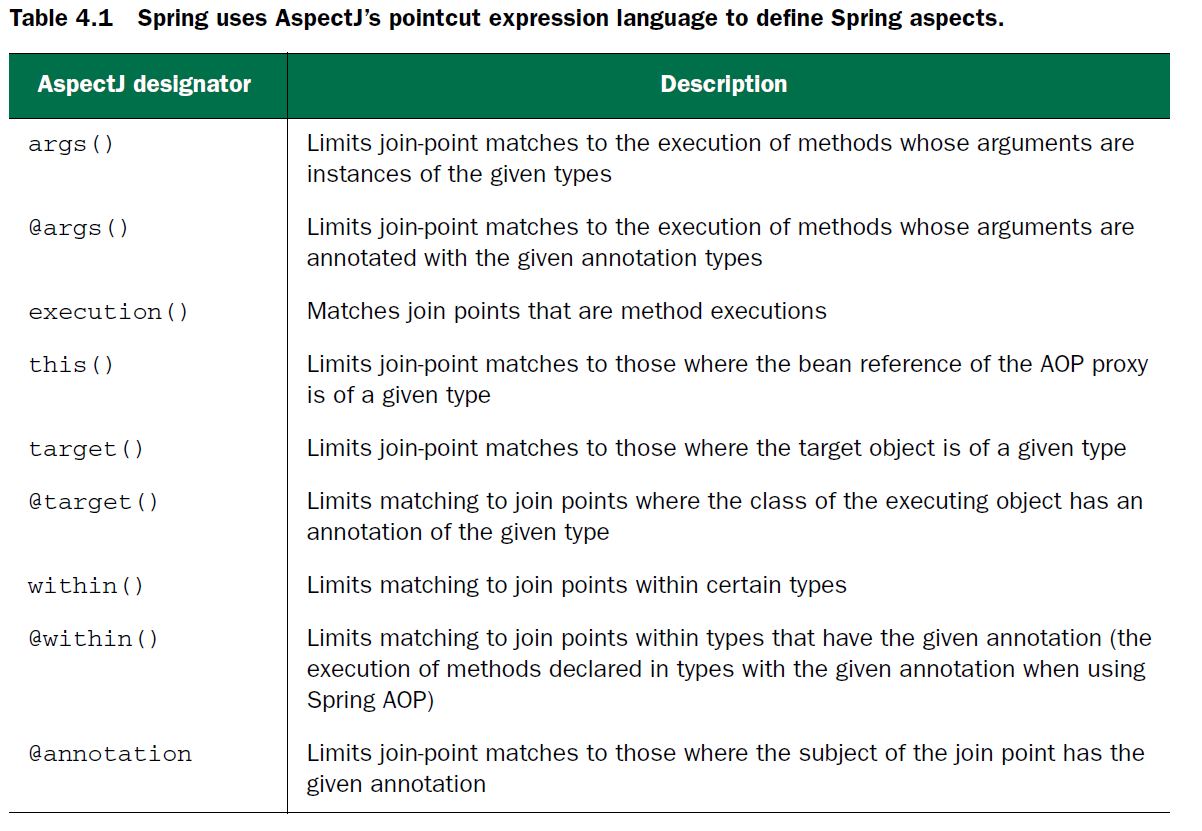
Spring’s support for AOP comes in four styles:

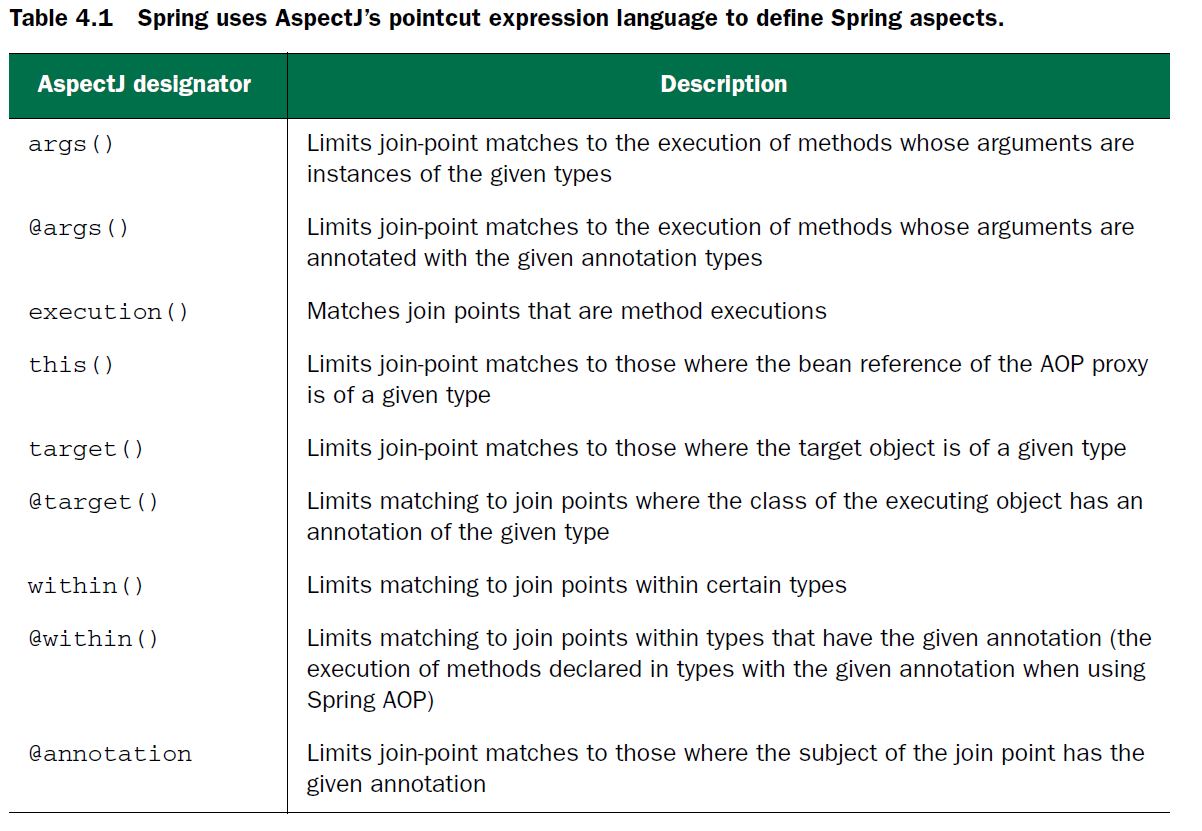
* Classic Spring proxy-based AOP
* Pure-POJO aspects
* @AspectJ annotation-driven aspects
* Injected AspectJ aspects (available in all versions of Spring)

***Selecting Join points with pointcuts:***

Pointcuts are used to pinpoint where an aspect’s advice should be applied.

Spring only supports a subset of the pointcut designators available in AspectJ. Table below lists the AspectJ pointcut designators that are supported in Spring AOP.





Attempting to use any of AspectJ’s other designators will result in an IllegalArgumentException being thrown.

Note that the execution designator is the only one that actually performs matches. The other designators are used to limit those matches. This means execution is the primary designator you’ll use in every pointcut definition you write. You’ll use the other designators to constrain the pointcut’s reach.

***Writing pointcuts:***

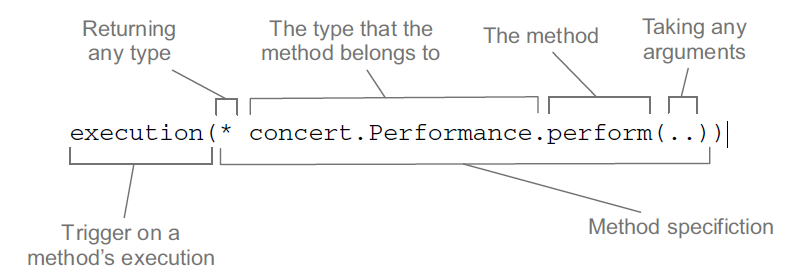
Let’s define a Performance interface:

package concert;

public interface Performance {

public void perform();

}

Performance represents any kind of live performance, such as a stage play, a movie, or a concert. Let’s say that you want to write an aspect that triggers off Performance’s perform() method. Below image shows a pointcut expression that can be used to apply advice whenever the perform() method is executed.

You use the execution() designator to select Performance’s perform() method. The method specification starts with an asterisk, which indicates that you don’t care what type the method returns. Then you specify the fully qualified class name and the name of the method you want to select. For the method’s parameter list, you use the double dot (..), indicating that the pointcut should select any perform() method, no matter what the argument list is.

You can limit the match by tacking on a within() designator. It confine the reach of that pointcut to only the concert package.

execution(\* concert.Performance.perform(..)) &&within(concert.\*))

the&&operator to combine the execution() and within() designators in an “and” relationship (where both designators must match for the pointcut to match). Similarly, you could use the || operator to indicate an “or” relationship. And the !operator can be used to negate the effect of a designator.

You’re free to use and in place of &&when specifying pointcuts in a Spring XML-based configuration. Likewise, or and not can be used in place of || and !, respectively.

***Selecting beans in pointcuts:***

Spring adds a bean() designator that lets you identify beans by their ID in a pointcut expression. bean() takes a bean ID or name as an argument and limits the pointcut’s effect to that specific bean.

For example, consider the following pointcut:

execution(\* concert.Performance.perform()) and bean('woodstock')

Using negation to apply an aspect to all beans that don’t have a specific ID:

execution(\* concert.Performance.perform()) and !bean('woodstock')

**Creating Annotated aspects:-**

***Defining an aspect:***

The following listing shows the Audience class that defines the aspect.

package concert;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class Audience {

@Before("execution(\*\* concert.Performance.perform(..))")

public void silenceCellPhones() {

System.out.println("Silencing cell phones");

}

@Before("execution(\*\* concert.Performance.perform(..))")

public void takeSeats() {

System.out.println("Taking seats");

}

@AfterReturning("execution(\*\* concert.Performance.perform(..))")

public void applause() {

System.out.println("CLAP CLAP CLAP!!!");

}

@AfterThrowing("execution(\*\* concert.Performance.perform(..))")

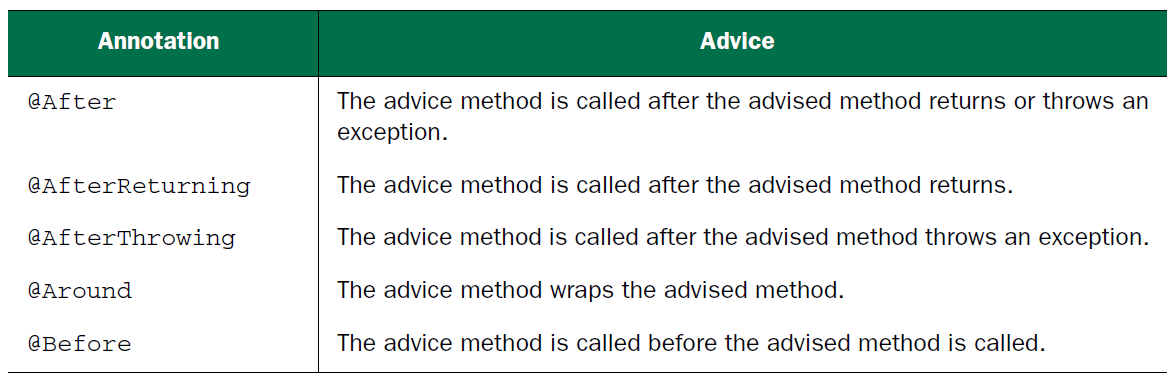
public void demandRefund() {

System.out.println("Demanding a refund");

}

}

The Audience class is annotated with @Aspect. This annotation indicates that Audience isn’t just any POJO—it’s an aspect. And throughout the Audience class are methods that are annotated to define the specifics of the aspect.

AspectJ provides five annotations for defining advice.

Problem:-

It’s a shame that you had to repeat that same pointcut expression(("execution(\*\* concert.Performance.perform(..))")) four times. Duplication like this doesn’t feel right.

Solution:-

Fortunately, there’s a way: the @Pointcut annotation defines a reusable pointcut within an@AspectJ aspect.

package concert;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class Audience {

@Pointcut("execution(\*\* concert.Performance.perform(..))")

public void performance() {}

@Before("performance()")

public void silenceCellPhones() {

System.out.println("Silencing cell phones");

}

@Before("performance()")

public void takeSeats() {

System.out.println("Taking seats");

}

@AfterReturning("performance()")

public void applause() {

System.out.println("CLAP CLAP CLAP!!!");

}

@AfterThrowing("performance()")

public void demandRefund() {

System.out.println("Demanding a refund");

}

}

And, just like any other Java class, Audience class can be wired as a bean in Spring Java config with @Bean. Also, if you’re using JavaConfig, you can turn on auto-proxying by applying the @EnableAspectJAutoProxy annotation at the class level of the configuration class. If @EnableAspectJAutoProxy is missed, then Audience would only be a regular bean and not aspect.

package concert;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration

@EnableAspectJAutoProxy

@ComponentScan

public class ConcertConfig {

@Bean

public Audience audience() {

return new Audience();

}

}

The XML configuration for same can be done as follows.

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

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

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

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

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

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

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

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

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

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

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

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

<aop:aspectj-autoproxy />

<bean class="concert.Audience" />

</beans>

***Creating Around advice:***

package concert;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class Audience {

@Pointcut("execution(\*\* concert.Performance.perform(..))")

public void performance() {}

@Around("performance()")

public void watchPerformance(ProceedingJoinPoint jp) {

try {

System.out.println("Silencing cell phones");

System.out.println("Taking seats");

jp.proceed();

System.out.println("CLAP CLAP CLAP!!!");

} catch (Throwable e) {

System.out.println("Demanding a refund");

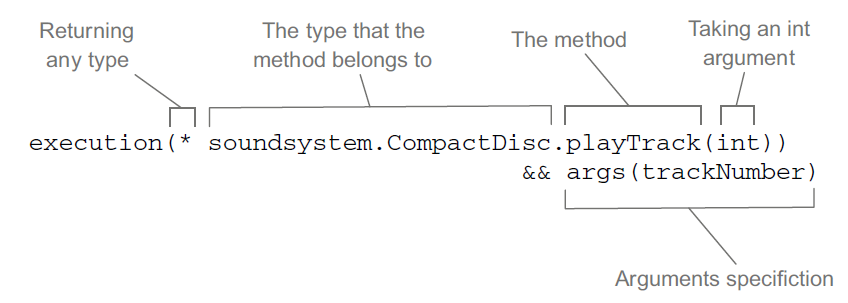
}

}

}

The first thing you’ll notice about this new advice method is that it’s given a ProceedingJoinPoint as a parameter. This object is necessary because it’s how you can invoke the advised method from within your advice. The advice method will do everything it needs to do; and when it’s ready to pass control to the advised method, it will call ProceedingJoinPoint’s proceed() method.

Note that it’s crucial that you remember to include a call to the proceed()method.



***Handling Parameters in advice:***

package soundsystem;

import java.util.HashMap;

import java.util.Map;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class TrackCounter {

private Map<Integer, Integer> trackCounts =new HashMap<Integer, Integer>();

@Pointcut("execution(\* soundsystem.CompactDisc.playTrack(int))&&args(trackNumber)")

public void trackPlayed(int trackNumber) {}

@Before("trackPlayed(trackNumber)")

public void countTrack(int trackNumber) {

int currentCount = getPlayCount(trackNumber);

trackCounts.put(trackNumber, currentCount + 1);

}

public int getPlayCount(int trackNumber) {

return trackCounts.containsKey(trackNumber)? trackCounts.get(trackNumber) : 0;

}

}

***Annotating Introductions:-***

An AOP concept known as *introduction*, aspects can attach new methods to Spring beans.

Recall that in Spring, aspects are proxies that implement the same interfaces as the beans they wrap. What if, in addition to implementing those interfaces, the proxy is also exposed through some new interface? Then any bean that’s advised by the aspect will appear to implement the new interface, even if its underlying implementation class doesn’t.

You want to introduce the following Encoreable interface.

package concert;

public interface Encoreable {

void performEncore();

}

package concert;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.DeclareParents;

@Aspect

public class EncoreableIntroducer {

@DeclareParents(value="concert.Performance+",

defaultImpl=DefaultEncoreable.class)

public static Encoreable encoreable;

}

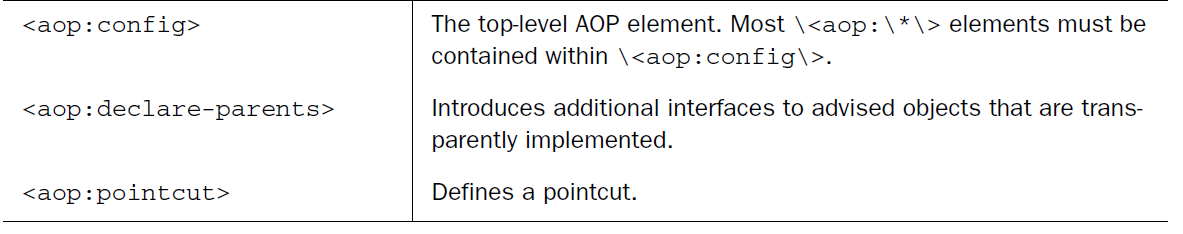
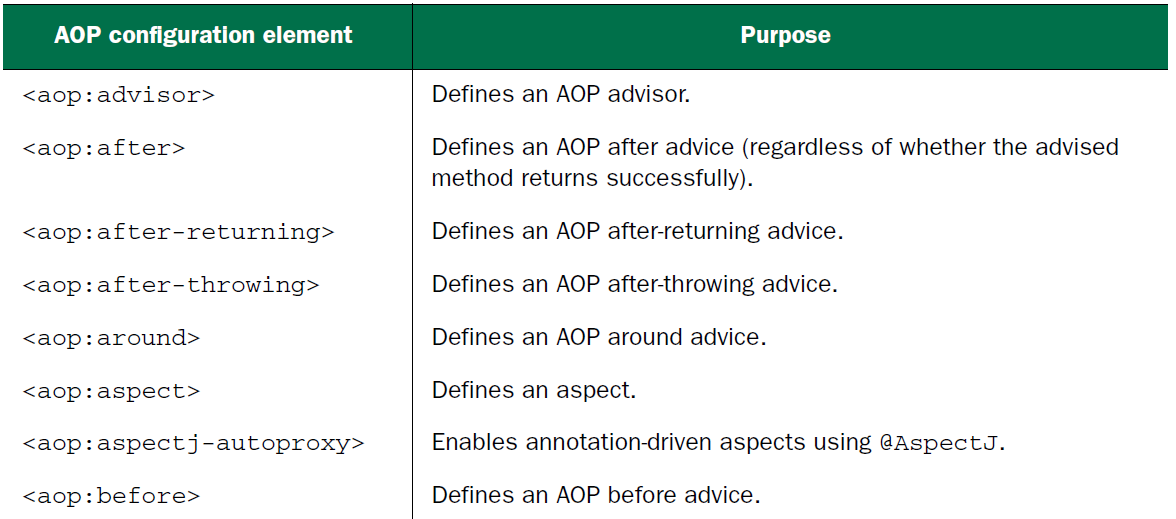
EncoreableIntroducer is an aspect. But unlike the aspects you’ve created so far, it doesn’t provide before, after, or around advice. Instead, it introduces the Encoreable interface to Performance beans using the @DeclareParents annotation. The @DeclareParents annotation is made up of three parts:

* The value attribute identifies the kinds of beans that should be introduced with the interface. In this case, that’s anything that implements the Performance interface. (The plus sign at the end specifies any subtype of Performance, as opposed to Performance itself.)
* The defaultImpl attribute identifies the class that will provide the implementation for the introduction. Here you’re saying that DefaultEncoreable will provide that implementation.
* The static property that is annotated by @DeclareParents specifies the interface that’s to be introduced. In this case, you’re introducing the Encoreable interface.

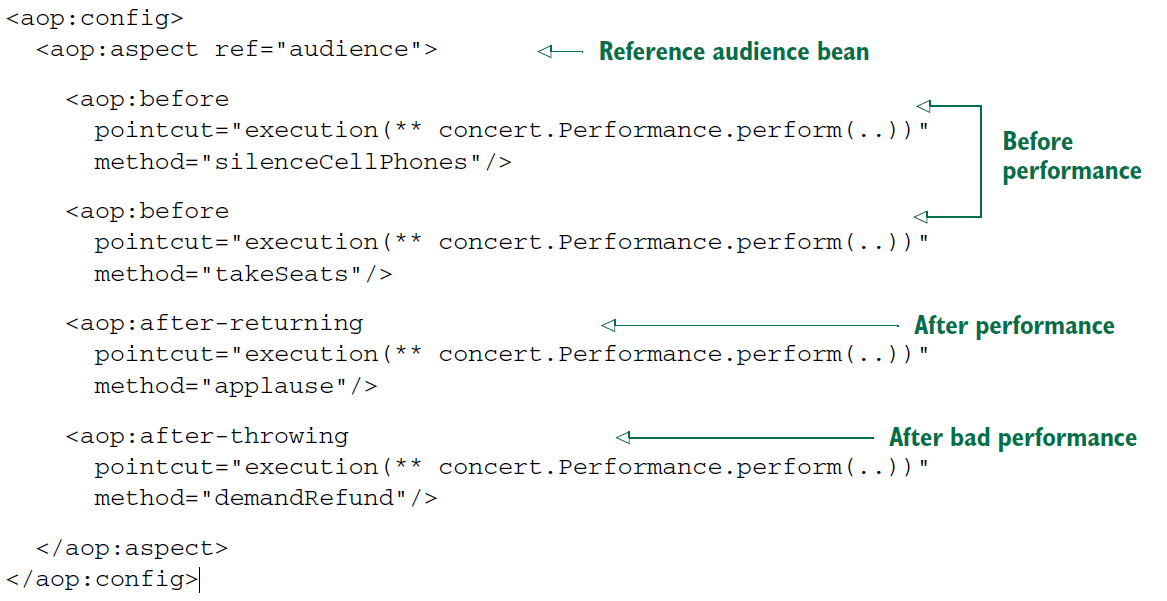
As with any aspect, you need to declare EncoreableIntroducer as a bean in the Spring application context:

<bean class="concert.EncoreableIntroducer" />

**Declaring aspect in XML:-**



***Declaring Before and After advice:-***

****package concert;

public class Audience {

public void silenceCellPhones() {

System.out.println("Silencing cell phones");

}

public void takeSeats() {

System.out.println("Taking seats");

}

public void applause() {

System.out.println("CLAP CLAP CLAP!!!");

}

public void demandRefund() {

System.out.println("Demanding a refund");

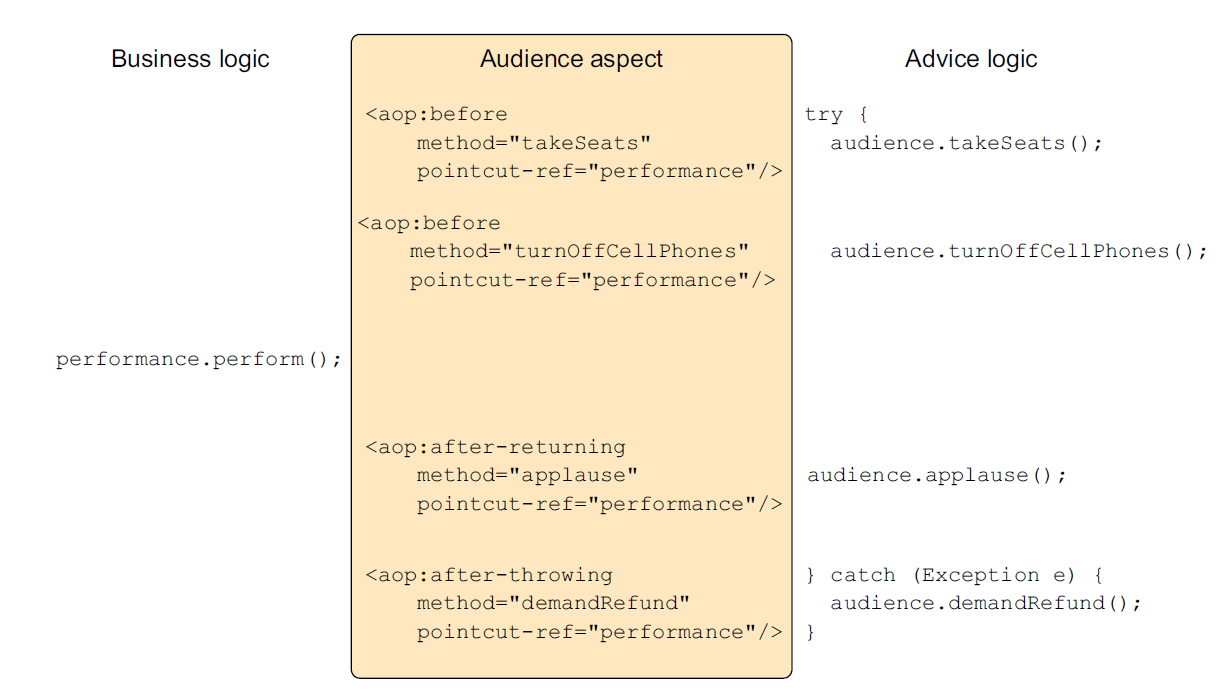
}

}

The first thing to notice about the Spring AOP configuration elements is that most of them must be used in the context of the <aop:config>element. There are a few exceptions to this rule, but when it comes to declaring beans as aspects, you’ll always start with <aop:config>.

In <aop:config>, you may declare one or more advisers, aspects, or pointcuts. Inlisting 4.9, you declare a single aspect using the <aop:aspect>element. The ref attributereferences the POJO bean that will be used to supply the functionality of theaspect—in this case, audience. The bean that’s referenced by the ref attribute willsupply the methods called by any advice in the aspect.

It’s worth noting that the referenced advice bean can be any type that providesmethods to be called at the designated pointcuts.



You’ve probably noticed that the value of the **pointcut**attribute is the same for all the advice elements. That’s because all the advice is being applied to the same pointcut. When you found the same kind of duplication in your AspectJ-annotated advice, you eliminated it by using the @Pointcut annotation.

<aop:config>

<aop:aspect ref="audience">

<aop:pointcut id="performance"

expression="execution(\*\* concert.Performance.perform(..))" />

<aop:before pointcut-ref="performance" method="silenceCellPhones"/>

<aop:before pointcut-ref="performance" method="takeSeats"/>

<aop:after-returning pointcut-ref="performance" method="applause"/>

<aop:after-throwing pointcut-ref="performance" method="demandRefund"/>

</aop:aspect>

</aop:config>

***Declaring Around advice:-***

package concert;

import org.aspectj.lang.ProceedingJoinPoint;

public class Audience {

public void watchPerformance(ProceedingJoinPoint jp) {

try {

System.out.println("Silencing cell phones");

System.out.println("Taking seats");

jp.proceed();

System.out.println("CLAP CLAP CLAP!!!");

} catch (Throwable e) {

System.out.println("Demanding a refund");

}

}

}

<aop:config>

<aop:aspect ref="audience">

<aop:pointcut id="performance"

expression="execution(\*\* concert.Performance.perform(..))" />

<aop:around pointcut-ref="performance" method="watchPerformance"/>

</aop:aspect>

</aop:config>

***Passing Parameter to an advice:-***

package soundsystem;

import java.util.HashMap;

import java.util.Map;

public class TrackCounter {

private Map<Integer, Integer> trackCounts =new HashMap<Integer, Integer>();

public void countTrack(int trackNumber) {

int currentCount = getPlayCount(trackNumber);

trackCounts.put(trackNumber, currentCount + 1);

}

public int getPlayCount(int trackNumber) {

return trackCounts.containsKey(trackNumber)? trackCounts.get(trackNumber) : 0;

}

}

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

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

<bean id="trackCounter" class="soundsystem.TrackCounter" />

<bean id="cd" class="soundsystem.BlankDisc">

<property name="title" value="Sgt. Pepper's Lonely Hearts Club Band" />

<property name="artist" value="The Beatles" />

<property name="tracks">

<list>

<value>Sgt. Pepper's Lonely Hearts Club Band</value>

<value>With a Little Help from My Friends</value>

<value>Lucy in the Sky with Diamonds</value>

<value>Getting Better</value>

<value>Fixing a Hole</value>

<!-- ...other tracks omitted for brevity... -->

</list>

</property>

</bean>

<aop:config>

<aop:aspect ref="trackCounter">

<aop:pointcut id="trackPlayed" expression=

"execution(\* soundsystem.CompactDisc.playTrack(int)) and args(trackNumber)" />

<aop:before pointcut-ref="trackPlayed" method="countTrack"/>

</aop:aspect>

</aop:config>

</beans>

***Introducing new functionality with aspect:***

AspectJ’s @DeclareParents annotation to magically introduce a new method into an advised bean. But AOP introductions aren’t exclusive to AspectJ. Using the <aop:declare-parents>element from

Spring’s aop namespace, you can do similar magic in XML. The following snippet of XML is equivalent to the AspectJ-based introduction you created earlier:

<aop:aspect>

<aop:declare-parents types-matching="concert.Performance+"

implement-interface="concert.Encoreable"

default-impl="concert.DefaultEncoreable"/>

</aop:aspect>

As its name implies, <aop:declare-parents>declares that the beans it advises will have new parents in its object hierarchy. Specifically, in this case you’re saying that the beans whose type matches the Performance interface (per the types-matching attribute) should have Encoreable in their parentage (per the implement-interface attribute). The final matter to settle is where the implementation of the Encoreable’s methods will come from.

There are two ways to identify the implementation of the introduced interface. In this case, you’re using the default-impl attribute to explicitly identify the implementation by its fully qualified class name. Alternatively, you could identify it using the delegate-ref attribute:

<aop:aspect>

<aop:declare-parents types-matching="concert.Performance+"

implement-interface="concert.Encoreable"

delegate-ref="encoreableDelegate"

</aop:aspect>

The delegate-ref attribute refers to a Spring bean as the introduction delegate. This assumes that a bean with an ID of encoreableDelegate exists in the Spring context:

<bean id="encoreableDelegate" class="concert.DefaultEncoreable" />

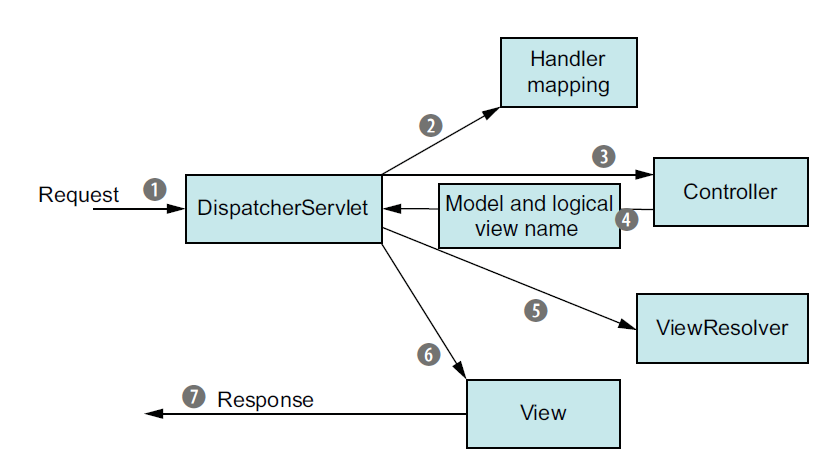
The difference between directly identifying the delegate using default-impl and indirectly using delegate-ref is that the latter will be a Spring bean that itself may be injected, advised, or otherwise configured through Spring.

***Injecting AspectJ aspects:-***

**Page # 125** of Spring in Action, 4th edition.

**Spring MVC**

**Following the life of a request:-**



The first stop in the request’s travels is at Spring’s**DispatcherServlet**. Like most Javabased web frameworks, Spring MVC funnels requests through a single front controller servlet. **A *front controller***is a common web application pattern where a single servlet delegates responsibility for a request to other components of an application to perform actual processing.

The **DispatcherServlet’s**job is to send the request on to a Spring MVC controller. A typical application may have several **controllers**, and DispatcherServlet needs some help deciding which controller to send the request to.

TheDispatcherServlet consults one or more **handler mappings**to figure out which **controller** the request’s next stop will be. The handler mapping pays particular attention to the URL carried by the request when making its decision.

Once an appropriate controller has been chosen, DispatcherServlet sends the request on its merry way to the chosen controller. At the controller, the request drops off its payload and patiently waits while the controller processes that information.

NOTE: - a well-designed controller performs little or no processing itself and instead delegates responsibility for the business logic to one or more service objects.

The logic performed by a controller often results in some information that needs to be carried back to the user and displayed in the browser. This information is referred to as the ***model***.

One of the last things a controller does is package up the **model** data and identifies the name of a view that should render the output. It then sends the request, along with the **model and logical view name**, back to the DispatcherServlet. View name only carries a logical name that will be used to look up the actualview that will produce the result.

The DispatcherServlet consults a **view resolver**to map the logical view name to a specific view implementation, which may or may not be a JSP.

Now that DispatcherServlet knows which **view**will render the result, the request’s job is almost over. Its final stop is at the **view** implementation, typically a JSP, where it delivers the model data.

**Setting up Spring MVC:-**

**Configuring DispatcherServlet using Java code:**

*DispatcherServlet* is the centerpiece of Spring MVC. It’s where the request first hits the framework, and it’s responsible for routing the request through all the other components.

package spittr.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class SpittrWebAppInitializer extends

AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected String[] getServletMappings() {

return new String[] { "/" };

}

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class<?>[] { RootConfig.class };

}

@Override

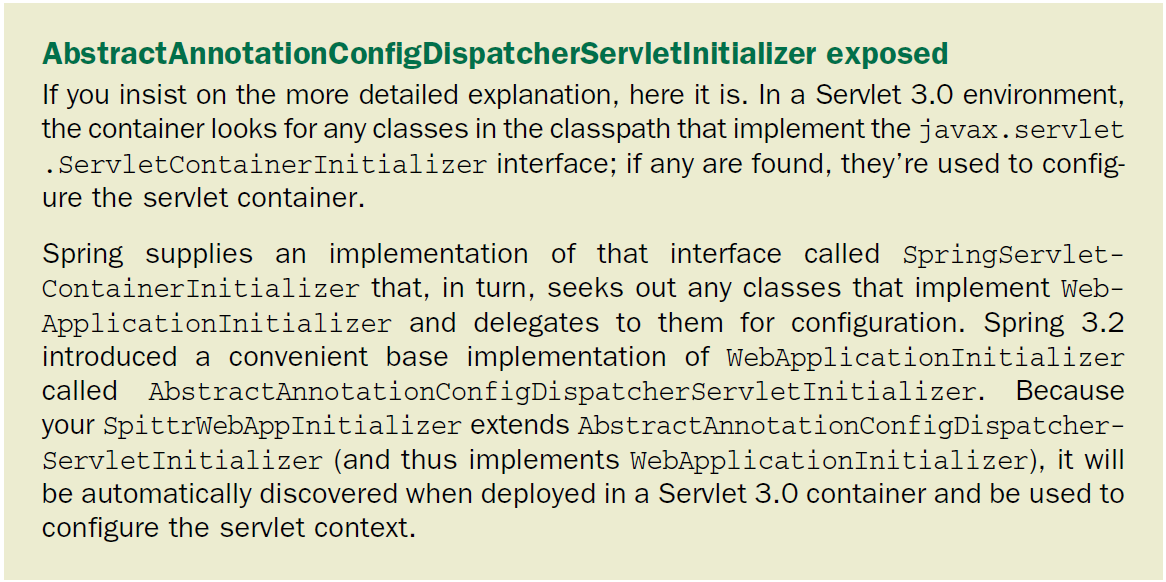
protected Class<?>[] getServletConfigClasses() {

return new Class<?>[] { WebConfig.class };

}

}

Any class that extends *AbstractAnnotationConfigDispatcherServletInitializer* will automatically be used to configure *DispatcherServlet* and the Spring application context in the application’s servlet context.



The *SpittrWebAppInitializer* overrides three methods.

The first method, *getServletMappings*(), identifies one or more paths that *DispatcherServlet* will be mapped to. In this case, it’s mapped to /, indicating that it will be the application’s default servlet. It will handle all requests coming into the application.

Under the covers, *AbstractAnnotationConfigDispatcherServletInitializer* creates both a *DispatcherServlet* and a *ContextLoaderListener*.

The @*Configuration* classes returned from *getServletConfigClasses*() will define beans for *Dispatcher*-*Servlet’s* application context.

Meanwhile, the @*Configuration* class’s returned *getRootConfigClasses*() will be used to configure the application context created by *ContextLoaderListener*.

In this case, your root configuration is defined in *RootConfig*, whereas *Dispatcher*-*Servlet’s* configuration is declared in *WebConfig*.

When *DispatcherServlet* starts up, it creates a Spring application context and starts loading it with beans declared in the configuration files or classes that it’s given.

With the *getServletConfigClasses*() method, you’ve asked that *Dispatcher*-*Servlet* load its application context with beans defined in the *WebConfig* configuration class (using Java configuration).

But in Spring web applications, there’s often another application context. This other application context is created by *ContextLoaderListener*.

Whereas *DispatcherServlet* is expected to load beans containing web components such as controllers, view resolvers, and handler mappings, *ContextLoaderListener* is expected to load the other beans in your application.

**Enabling Spring MVC:**

The *very* simplest Spring MVC configuration you can create is a class annotated with *@EnableWebMvc*:

package spittr.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration

@EnableWebMvc

public class WebConfig {

}

This will work, and it will enable Spring MVC.

But it leaves a lot to be desired:

 No view resolver is configured. As such, Spring will default to using *BeanNameViewResolver*, a view resolver that resolves views by looking for beans whose ID matches the view name and whose class implements the *View* interface.

 Component-scanning isn’t enabled. Consequently, the only way Spring will find any controllers is if you declare them explicitly in the configuration.

 As it is, *DispatcherServlet* is mapped as the default servlet for the application and will handle *all* requests, including requests for static resources, such as images and stylesheets (which is probably not what you want in most cases).

The new *WebConfig* in the next listing addresses these concerns.

package spittr.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.ViewResolver;

import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration

@EnableWebMvc

@ComponentScan("spitter.web")

public class WebConfig extends WebMvcConfigurerAdapter {

@Bean

public ViewResolver viewResolver() {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

resolver.setExposeContextBeansAsAttributes(true);

return resolver;

}

@Override

public void configureDefaultServletHandling(

DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

}

*WebConfig* is annotated with @*ComponentScan* so that the *spitter*.*web* package will be scanned for components. @*Controller* will make them candidates for component-scanning.

Consequently, you won’t have to explicitly declare any controllers in the configuration class.

Next, you add a *ViewResolver* bean. More specifically, it’s an *Internal*-*ResourceViewResolver*. For now, just know that it’s configured to look for JSP files by wrapping view names with a specific prefix and suffix.

Finally, Finally, this new *WebConfig* class extends *WebMvcConfigurerAdapter* and overrides its *configureDefaultServletHandling*() method. By calling *enable*() on the given *DefaultServletHandlerConfigurer*, you’re asking *DispatcherServlet* to forward requests for static resources to the servlet container’s default servlet and not to try to handle them itself.

*RootConfig will be covered later:-*

**Writing a simple Controller:-**

package spittr.web;

import static org.springframework.web.bind.annotation.RequestMethod.\*;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

public class HomeController {

@RequestMapping(value="/", method=GET)

public String home() {

return "home";

}

}

The first thing you’ll notice about *HomeController* is that it’s annotated with @*Controller*. Although it’s clear that this annotation declares a controller, the annotation has little to do with Spring MVC.

@*Controller* is a stereotype annotation, based on the @*Component* annotation. Its purpose here is entirely for the benefit of component-scanning.

*HomeController’s* only method, the *home*() method, is annotated with @*Request*-*Mapping*. The *value* attribute specifies the request path that this method will handle, and the *method* attribute details the HTTP method that it can handle. In this case, whenever an HTTP *GET* request comes in for /, the *home*() method will be called.

This *String* will be interpreted by Spring MVC as the name of the view that will be rendered. *DispatcherServlet* will ask the view resolver to resolve this logical view name into an actual view.

Given the way you configured *InternalResourceViewResolver*, the view name “*home*” will be resolved as a JSP at /WEB-INF/views/home.jsp.

**Defining class-level request handling:-**

package spittr.web;

import static org.springframework.web.bind.annotation.RequestMethod.\*;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

@RequestMapping("/")

public class HomeController {

@RequestMapping(method=GET)

public String home() {

return "home";

}

}

Any time there’s a class-level @*RequestMapping* on a controller class, it applies to all handler methods in the controller. Then any @*RequestMapping* annotations on handler methods will complement the class-level @*RequestMapping*.

In the case of *HomeController*, there’s only one handler method. Its @*RequestMapping*, when taken together with the class-level @*RequestMapping*, indicates that the *home*() method will handle *GET* requests for /.

The *value* attribute of @*RequestMapping* accepts an array of *String*. So far, you’ve only given it a single *String* value of “/”. But you can also map it to requests whose path is /homepage by changing the class-level

@*RequestMapping* to look like this:

@Controller

@RequestMapping({"/", "/homepage"})

public class HomeController {

...

}

**Passing model data to the view:-**

package spittr.web;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import spittr.Spittle;

import spittr.data.SpittleRepository;

@Controller

@RequestMapping("/spittles")

public class SpittleController {

private SpittleRepository spittleRepository;

@Autowired

public SpittleController(SpittleRepository spittleRepository) {

this.spittleRepository = spittleRepository;

}

@RequestMapping(method=RequestMethod.GET)

public String spittles(Model model) {

model.**addAttribute**("spittleList",spittleRepository.findSpittles(Long.MAX\_VALUE, 20));

return "spittles";

}

}

*SpittleController* has a constructor that’s annotated with @*Autowired* to be given a SpittleRepository. That *SpittleRepository* is then used in the *spittles*() method to fetch a list of recent spittles.

Notice that the *spittles*() method is given a *Model* as a parameter. This is so that *spittles*() can populate the model with the *Spittle* list it retrieves from the repository.

The *Model* is essentially a map (that is, a collection of key-value pairs) that will be handed off to the view so that the data can be rendered to the client. When *addAttribute*() is called without specifying a key, the key is inferred from the type of object being set as the value. In this case, because it’s a *List*<*Spittle*>, the key will be inferred as *spittleList*.

Likewise, if you’d prefer to work with a non-Spring type, you can ask for a *java.util.Map* instead of *Model*. Here’s another version of *spittles*() that’s functionallyequivalent to the others:

@RequestMapping(method=RequestMethod.GET)

public String spittles(Map model) {

model.**put**("spittleList",spittleRepository.findSpittles(Long.MAX\_VALUE, 20));

return "spittles";

}

Alternatively, here’s another way to write the *spittles*() method:

@RequestMapping(method=RequestMethod.GET)

public List<Spittle> spittles() {

return spittleRepository.findSpittles(Long.MAX\_VALUE, 20));

}

Which is same as below

@RequestMapping(method=RequestMethod.GET)

public String spittles(Model model) {

model.**addAttribute**(spittleRepository.findSpittles(Long.MAX\_VALUE, 20));

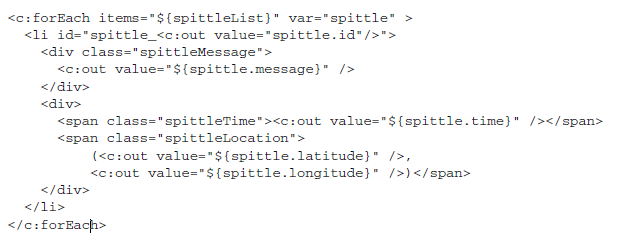
return "spittles";

}

When a handler method returns an object or a collection like this, the value returned is put into the model, and the model key is inferred from its type.

As for the logical view name, it’s inferred from the request path. Because this method handles *GET* requests for */spittles*, the view name is *spittles* (chopping off the leading slash).

Whichever way you choose, a list of *Spittle* objects is stored in the model with a key of *spittleList* and given to the view whose name is *spittles*.

JSP view:-

**Accepting request input:-**

Spring MVC provides several ways that a client can pass data into a controller’s handler method. These include

 Query parameters

 Form parameters

 Path variables

**Taking query parameters:-**

Request - xxx/spittles?max=238900&count=50

@RequestMapping(method=RequestMethod.GET)

public List<Spittle> spittles(

@RequestParam("max") long max,

@RequestParam("count") int count){

return spittleRepository.findSpittles(max, count);

}

@*RequestParam* will help you get the query parameter in the implementation.

Now, if you want default value in no parameter is passed.

@RequestMapping(method=RequestMethod.GET)

public List<Spittle> spittles(

@RequestParam(value="max",defaultValue=MAX\_LONG\_AS\_STRING) long max,

@RequestParam(value="count", defaultValue="20") int count) {

return spittleRepository.findSpittles(max, count);

}

***Note:*** - Value for *defaultValue* attribute should be string.

**Taking input via path parameters:-**

Let’s say your application needs to support the display of single *Spittle*, given its ID.As a general rule, query parameters should not be used to identify a resource. A *GET* request for */spittles/12345* is better than one for */spittles/show?spittle\_id=12345*.

Spring MVC allows for placeholders in an @*RequestMapping* path. The placeholders are names surrounded by curly braces ({ and }).

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

public String spittle(@PathVariable("spittleId") long spittleId,Model model) {

model.addAttribute(spittleRepository.findOne(spittleId));

return "spittle";

}

If the request is a *GET* request for /spittles/54321, then 54321 will be passed in as the value of *spittleId*.

Because the method parameter name happens to be the same as the placeholder name, you can optionally omit the *value* parameter on @*PathVariable*: If no *value* attribute is given for @*PathVariable*, it assumes the placeholder’s name is the same as the method parameter name.

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

public String spittle(@PathVariable long spittleId, Model model) {

model.addAttribute(spittleRepository.findOne(spittleId));

return "spittle";

}

**Processing forms:-**

There are two sides to working with forms: displaying the form and processing the data the user submits from the form.

package spittr.web;

import static org.springframework.web.bind.annotation.RequestMethod.\*;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import spittr.Spitter;

import spittr.data.SpitterRepository;

@Controller

@RequestMapping("/spitter")

public class SpitterController {

@RequestMapping(value="/register", method=GET)

public String showRegistrationForm() {

return "registerForm";

}

}

The *showRegistrationForm*() method’s @*RequestMapping* annotation, along with the class-level @*RequestMapping* annotation, declares that it will handle HTTP *GET* requests for */spitter/register*. Given how you’ve configured *InternalResourceViewResolver*, that means the JSP at /WEB-INF/views/registerForm.jsp will be called on to render the registration form.

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ page session="false" %>

<html>

<head>

<title>Spittr</title>

<link rel="stylesheet" type="text/css" href="<c:url value="/resource/style.css" />" >

</head>

<body>

<h1>Register</h1>

<form method="POST">

First Name: <input type="text" name="firstName" />

<br />

Last Name: <input type="text" name="lastName" />

<br />

Username: <input type="text" name="username" />

<br />

Password: <input type="password" name="password" />

<br />

<input type="submit" value="Register" />

</form>

</body>

</html>

***Notice*** that the *<form>* tag **doesn’t have an *action***parameter set. Because of that, when this form is submitted, it will be posted back to the same URL path that displayed it.

package spittr.web;

import static org.springframework.web.bind.annotation.RequestMethod.\*;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import spittr.Spitter;

import spittr.data.SpitterRepository;

@Controller

@RequestMapping("/spitter")

public class SpitterController {

private SpitterRepository spitterRepository;

@Autowired

public SpitterController(SpitterRepository spitterRepository) {

this.spitterRepository = spitterRepository;

}

@RequestMapping(value="/register", method=GET)

public String showRegistrationForm() {

return "registerForm";

}

@RequestMapping(value="/register", method=POST)

public String processRegistration(Spitter spitter) {

spitterRepository.save(spitter);

return "redirect:/spitter/" + spitter.getUsername();

}

@RequestMapping(value="/{username}", method=GET)

public String showSpitterProfile(@PathVariable String username, Model model) {

Spitter spitter = spitterRepository.findByUsername(username);

model.addAttribute(spitter);

return "profile";

}

}

Notice the new *processRegistration*() method: it’s given a *Spitter* object as a parameter. This object has *firstName*, *lastName*, *username*, and *password* properties that will be populated from the request parameters of the same name.

The last thing that *processRegistration*() does is return a *String* specifying the view.

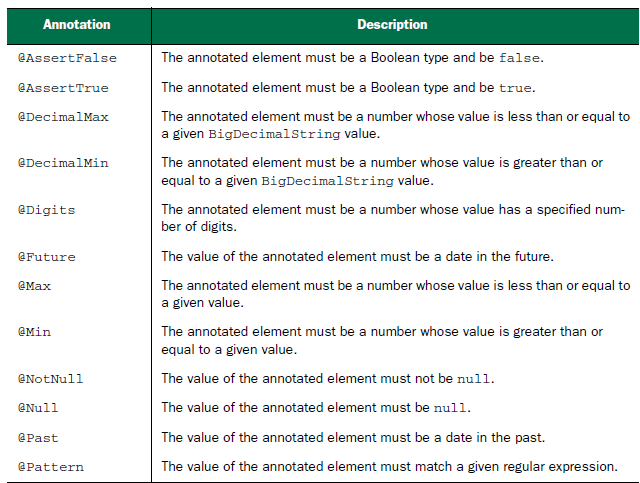
When *InternalResourceViewResolver* sees the ***redirect***: prefix on the view specification, it knows to interpret it as a redirect specification instead of as a view name.

It’s worth noting that in addition to *redirect*:,*InternalResourceViewResolver* also recognizes the *forward*: prefix.

**Validating forms:-**

Starting with Spring 3.0, Spring supports the Java Validation API in Spring MVC. No extra configuration is required to make Java Validation work in Spring MVC. You just need to make sure an implementation of the Java API, such as Hibernate Validator, is in the project’s classpath.

The Java Validation API defines several annotations that you can put on properties to place constraints on the values of those properties.



package spittr;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

public class Spitter {

private Long id;

@NotNull

@Size(min=5, max=16)

private String username;

@NotNull

@Size(min=5, max=25)

private String password;

@NotNull

@Size(min=2, max=30)

private String firstName;

@NotNull

@Size(min=2, max=30)

private String lastName;

...}

Now that you have annotated *Spitter* with validation constraints, you need to change the *processRegistration*() method to apply validation. The new validation enabled *processRegistration*() is shown next.

@RequestMapping(value="/register", method=POST)

public String processRegistration(@Valid Spitter spitter,Errors errors) {

if (errors.hasErrors()) {

return "registerForm";

}

spitterRepository.save(spitter);

return "redirect:/spitter/" + spitter.getUsername();

}

The *Spitter* parameter is now annotated with @*Valid* to indicate to Spring that the command object has validation constraints that should be enforced.

If there are any validation errors, they’re available in the *Errors* object that you’re now asking for as a parameter to *processRegistration*(). (Note that it’s important that the *Errors* parameter immediately follow the @*Valid*-annotated parameter that’s being validated.)

If there are errors, Errors.hasErrors() returns registerForm, the view name for the registration form. This will take the user’s browser back to the registration form so they can correct any problems and try again. For now, the blank form will be displayed, but in the next chapter, you’ll adapt the form to show the values that were originally submitted and communicate validation problems to the user.

**Rendering web views**

**Understanding view resolution:-**

Spring MVC defines an interface named *ViewResolver* that looks a little something like this:

public interface ViewResolver {

View resolveViewName(String viewName, Locale locale)throws Exception;

}

The *resolveViewName*() method, when given a view name and a *Locale*, returns a *View* instance. *View* is another interface that looks like this:

public interface View {

String getContentType();

void render(Map<String, ?> model,HttpServletRequest request,

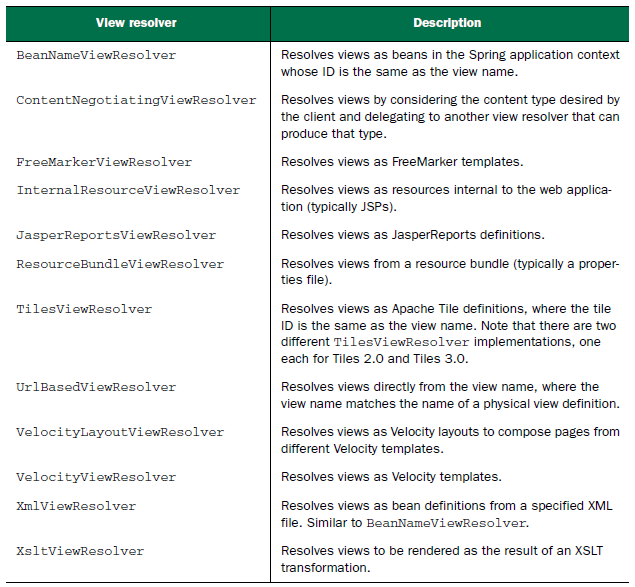
HttpServletResponse response) throws Exception;

}

The *View* interface’s job is to take the model, as well as the servlet request and response objects, and render output into the response.

Although you can write your own custom implementations of *ViewResolver* and *View*, and although there are some special cases where that’s necessary, typically you needn’t worry yourself with these interfaces.

Spring provides several out-of-the-box implementations

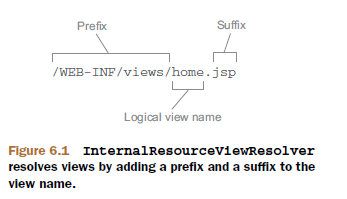


**JSP View Reslover:-**

Spring supports JSP views in two ways:

* *InternalResourceViewResolver* can be used to resolve view names into JSP files. Moreover, if you’re using JavaServer Pages Standard Tag Library (JSTL) tags in your JSP pages, *InternalResourceViewResolver* can resolve view names into JSP files fronted by *JstlView* to expose JSTL locale and resource bundle variables to JSTL’s formatting and message tags.
* Spring provides two JSP tag libraries, one for form-to-model binding and one providing general utility features.

**Configuring a JSP-ready view resolver:-**

*ResourceBundleViewResolver*, directly map a logical view name to a specific implementation of the View interface, *InternalResourceViewResolver* takes a more indirect approach.

You can configure *InternalResourceViewResolver* to apply this convention when resolving views by configuring it with this @*Bean*-annotated method:

@Bean

public ViewResolver viewResolver() {

InternalResourceViewResolver resolver =new InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

return resolver;

}

Optionally, if you prefer to use Spring’s XML-based configuration, you can configure

*InternalResourceViewResolver* like this:

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"

p:prefix="/WEB-INF/views/" p:suffix=".jsp" />

When a logical view name has a slash in it, that slash is carried over into the resource path name. Therefore, it maps to a JSP file that’s in a subdirectory of whatever directory is referenced by the *prefix* property.

if those JSP files are using JSTL tags for formatting or messages, then you may want to configure *InternalResourceViewResolver* to resolve a *JstlView* instead.

**Advanced Spring MVC:-**

**Alternate Spring MVC configuration:-**

Setting up Spring MVC by extending AbstractAnnotationConfigDispatcherServletInitializer class assumes that you want a basic DispatcherServlet and ContextLoaderListener setup and that your Spring configuration will be in Java instead of XML.

Fortunately, there are several ways that Spring returns some control to you when the garden-variety AbstractAnnotationConfigDispatcherServletInitializer configuration doesn’t fit your needs.

1. Customizing DispatcherServlet configuration

The three methods you wrote in SpittrWebAppInitializer were the only abstract ones you were required to override. But there are more methods that can be overridden to apply additional configuration.

One such method is customizeRegistration(). After AbstractAnnotationConfigDispatcherServletInitializer registers DispatcherServlet with the servlet container, it calls the customizeRegistration() method, passing in the ServletRegistration.Dynamic that resulted from the servlet registration. By overriding customizeRegistration(), you can apply additional configuration to DispatcherServlet.

If you plan to use Servlet 3.0 support for multipart configuration, you need to enable DispatcherServlet’s registration to enable multipart requests. You can override the customizeRegistration()method to set a MultipartConfigElement like this:

@Override

protected void customizeRegistration(Dynamic registration) {

registration.setMultipartConfig(

new MultipartConfigElement("/tmp/spittr/uploads"));

}

With the ServletRegistration.Dynamic that’s given to customizeRegistration(), you can do several things, including set the load-on-startup priority by calling setLoadOnStartup(), set an initialization parameter by calling setInitParameter(), and call setMultipartConfig() to configure Servlet 3.0 multipart support.

1. Adding additional servlets and filters

One of the nice things about working with a Java-based initializer is that (unlike with web.xml) you can define as many initializer classes as you want. Therefore, if you need to register any additional components into the web container, you need only create a new initializer class. The easiest way to do this is by implementing Spring’s WebApplicationInitializer interface.

The following listing shows how to create an implementation of WebApplicationInitializer that registers a servlet.

package com.myapp.config;

import javax.servlet.ServletContext;

import javax.servlet.ServletException;

import javax.servlet.ServletRegistration.Dynamic;

import org.springframework.web.WebApplicationInitializer;

import com.myapp.MyServlet;

public class MyServletInitializer implements WebApplicationInitializer {

@Override

public void onStartup(ServletContext servletContext)

throws ServletException {

Dynamic myServlet =

servletContext.addServlet("myServlet", MyServlet.class);

myServlet.addMapping("/custom/\*\*");

}

}

Above is a rather basic servlet-registering initializer class. It registers a servlet and maps it to a single path. You could use this approach to register DispatcherServlet manually.

Similarly, you can register listeners and filters by creating a new implementation of WebApplicationInitializer.

@Override

public void onStartup(ServletContext servletContext) throws ServletException {

javax.servlet.FilterRegistration.Dynamic filter =

servletContext.addFilter("myFilter", MyFilter.class);

filter.addMappingForUrlPatterns(null, false, "/custom/\*");

}

WebApplicationInitializer is a fine general-purpose way of registering servlets, filters, and listeners in Java when deploying to a Servlet 3.0 container.

But if you’re registering a filter and only need to map that filter to DispatcherServlet, then there’s a shortcut in AbstractAnnotationConfigDispatcherServletInitializer.

the following getServletFilters()method overrides the one from AbstractAnnotationConfigDispatcherServletInitializer to register a filter:

@Override

protected Filter[] getServletFilters() {

return new Filter[] { new MyFilter() };

}

Here it only returns a single filter, but it could return as many filters as you need. There’s no need to declare the mapping for the filters; any filter returned from getServletFilters() will automatically be mapped to DispatcherServlet.

1. Declaring DispatcherServlet in web.xml

Here’s a basic web.xml file with a typical setup for DispatcherServlet and ContextLoaderListener.

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

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.5" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<!- - Set root context location -- >

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

<listener>

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

</listener>

<servlet>

<servlet-name>appServlet</servlet-name>

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

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

If you’d rather specify the location of the DispatcherServlet configuration file, you can set a contextConfigLocation initialization parameter on the servlet. For example, the following DispatcherServlet configuration has DispatcherServlet loading its beans from /WEB-INF/spring/appServlet/servlet-context.xml:

<servlet>

<servlet-name>appServlet</servlet-name>

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

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

To load Java spring configuration files from web.xml, you’ll need to set up Spring MVC to load the configuration from @Configuration-annotated classes. To use Java-based configuration in Spring MVC, you need to tell DispatcherServlet and ContextLoaderListener to use AnnotationConfigWebApplicationContext, an implementation of WebApplicationContext that loads Java configuration classes instead of XML.

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

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.5" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<context-param>

<param-name>contextClass</param-name>

<param-value> org.springframework.web.context.support.AnnotationConfigWebApplicationContext </param-value>

</context-param>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.habuma.spitter.config.RootConfig</param-value>

</context-param>

<listener>

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

</listener>

<servlet>

<servlet-name>appServlet</servlet-name>

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

<init-param>

<param-name>contextClass</param-name>

<param-value> org.springframework.web.context.support.AnnotationConfigWebApplicationContext </param-value>

</init-param>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.habuma.spitter.config.WebConfigConfig</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

**Extra Notes –**

**Spring Bean factory implementation:**

BeanFactory bean = new XmlBeanFactory(new FileSystemResource(“spring.xml”));// Now all the objects are created

MyFirstClassobj = (MyFirstClass) Bean.getBean(“myFirstClass);

**Spring Application Context implementation:**

ApplicationContext context = new ClassPathApplicationContext (“spring.xml”);

MyFirstClassobj = (MyFirstClass) Bean.getBean(“myFirstClass);

Note: we are not using new FileSystemResource.

Here the spring.xml should be under the classpath i.e. under org.abc.**src**

**DI using a constructor**

1 <constructor-args value=”ABC”>

This will call a constructor which takes String as param

2 <constructor-args type=”java.lang.String” value=”20”>

3 <constructor-args type=”int” value=”20”>

4 <constructor-args index=”0” value=”ABC”>

<constructor-args index=”0” value=”20”>

**Inner Beans**

Inner bean is used when we are injecting an object in a particular bean def only.

<bean id=”abc” calss=”..Abc”>

<property name=”a” ref=”mno”/>

<property name=”b”>

**<bean class=”xyz”>**

**<property test=”123”/>**

**</>**

</></>

<bean id=”mno” class=”xyz”>

<property test=”123”/>

</>

Not in the above example property **b**is injected with a inner bean called **xyz.** Here xyz is use full only for the class abc and no other class wants so it has been defined with in that class. The bean mno can be used by some other class so it is defined normally and ref in the abc bean. Also note that the inner bean do not have ID (not mandatory).

**Alias**

Alas is another name**given** to out bean id

<alias name=”abc” alias=”new.name.abc”>

We can also give alias by using name

<bean id=”abc” calss=”..Abc” name=”new.abc”>

**Spring Collections**

<property name=”myList”>

<List>

<ref bean=”Obj1”/>

<value>Obj2</>

</>

</>

**Spring Collections**

<property name=”myMap”>

<map>

<entry key=”1” value=”A”/>

<entry key=”2” value=”B”/>

</>

</>

**Autowire**

**ByName**

Class ABC

Private MyClass**myClassA**;

Private MyClass**myClassB**;

Get() set()

<bean id=”abc” class=”..ABC” **autowire**=”byName” />

<bean id=”**myClassA**” calss=”MyClass”/>

<bean id=”**myClassB**” calss=”MyClass”/>

When the name of the property name in java matches the bean id name in xml then auto wire byname is performed.

**ByType**

Class ABC

Private **MyClass**myClassA;

Get() set()

<bean id=”abc” class=”..ABC” **autowire**=”byType” />

<bean id=”myClassA” calss=”**MyClass**”/>

Here auto wire will be done for the type of the property defined in the java class and the type in the xml. But if we have more than one bean def of the same class type then the auto wire by Type will not work

<bean id=”myClassA” calss=”**MyClass**”/>

<bean id=”myClassB” calss=”**MyClass**”/>

**Scope**

**Basic scope**

**<bean id=”” class=”” scope=”prototype”/>**

**Singleton:** This is the default scope. Only one object is created inside a particular spring container (created at the time of ApplicationContectObj is created) and server for all the requests and ref

**Prototype:**New bean created for every request and ref (by default no beans are created when the ApplicationContextobj is created)

**Web-Aware context scope:**This is when the spring is tied with web application

**Request:** for a new HTTP requests

**Session:** for every New HTTP Session

**Aware Interface**

**ApplicationContextAware – setApplicationContext(ApplicationContext context)**

**BeanNameAware – setBeanName(String beanName)**

**ApplicationEventPublisherAware ??????**

**BeanClassLoaderAware ????**

The aware interface is used to make our class aware or access to some of the important objects of spring.

If we want an application context in our particular class I can implement ApplicationContextAware interface and implement the method setApplicationContext(ApplicationContext context)

Class MyClass implements ApplicationContextAware {

Private ApplicationContext context = null;

Public void setApplicationContext(ApplicationContext context) {

This.context = context;

}

}

In the above code spring will set the context value in the setApplicationContext method. Now we can use the context obj for our further use foe Ex to load a bean context.getBean(“myBean”);

**Bean Definition Inheritance**

Bean inheritance is used to configure bean properties which are common for other bean definitions.

Ex:

<bean id=”myParentBean” class=”..MyBaseClass” >

<property name=”a” value=”A”/>

</>

<bean id=”myClass\_1” class=”..MyClass” **parent=” myParentBean”>**

<property name=”b” value=”B”/>

<property name=”c” value=”C”/>

</>

<bean id=”myClass\_2” class=”..MyClass” ” **parent=” myParentBean”**>

<property name=”c” value=”C”/>

</>

Class MyClass{

Private String a;

Private String b;

Private String c;

}

Context.getBean(“myClass\_1”);

//This will have the value set by both parent and child bean i.e. a,b,c

Context.getBean(“myClass\_2”);

//Here the value for “b” will be null because the value for “a” is set by parent and fro “c” set by child.

Note we can also override the parent property

Ex:

<bean id=”myClass\_1” class=”..MyClass” parent=”myParentBean”>

<property name=”a” value=”new\_A”/>

<property name=”b” value=”B”/>

<property name=”c” value=”C”/>

</>

Here the parent property value “A” will be overridden by “new\_A”

Note in Spring inheritance we can also inherit a Collection and we can also merge a parent collection with child list

Ex:

Class MyClass{

List<String>myList;

}

<bean id=”myParentBean” class=”..MyBaseClass” >

<property name=”myList”>

<list>

<value>ABC</>

</>

</>

</>

<bean id=”myClass\_1” class=”..MyClass” **parent=” myParentBean”**>

<property name=”myList”>

<list **merge=”true”**>

<value>ABC</>

</>

</></>

**Making a parent as template**

If don’t want to create a object of our parent class we can make if as abstract

<bean id=”myParentBean” class=”..MyBaseClass” **abstract=”true”**>

<property name=”a” value=”A”/>

</>

**Custom init() and destroy() mothods**

Class MyClass{

myInit(){

syso(“my init”);

}

myDestroy(){

syso(“my cleanup method”);

}

}

<bean id=”myParentBean” class=”..MyBaseClass” init-method=”myInit” destroy=”myDestroy”>

<property name=”a” value=”A”/>

</>

**Global Init and destroy**

<beans **default-init-method=”myInit” default-mestroy-method=”myDestroy”**>

<bean id=”myParentBean” class=”..MyBaseClass” >

<property name=”a” value=”A”/>

</>

</>

Here any class which has methods by the name myInit() and myDestroy() will get executed.

**Bean Post Processor**

Are class that tells spring to do some processing after initializing each and every bean. Note the bean post processer works for all the objects in the container and not on a particular object.

Class MyBeanPosProcessor implements **BeanPostProcessers** {

Public Object **postProcessAfterInitializition**(Object bean, String beanName) throws BeanException{

Syso(“After initialization of bean”);

Return bean;

}

Public Object **postProcessBeforeInitializition**(Object bean, String beanName) throws BeanException{

Syso(“Before initialization of bean”);

Return bean;

}

Note here we can also change the value of the bean object before returning it.

<bean calss=”..MyBeanPostProssor”>

Now MyBeanPostProcessor will work on all the object created by spring.

Note there is no destroy in BeanPostProcessor.

**Bean Factory Post Processor**

If we want to execute some code when bean factory object is initialized.

Class MyBeanFactoryPosProcessor implements **BeanFactoryPostProcessers** {

Public Object **postProcessBeanFactory**(ConfigurableListableBeanFactorybeanFactory) throws BeanException{

Syso(“My bean factory”);

Return bean;

}}

<bean calss=”..MyBeanFactoryPostProssor”>

This will work when both Bean factory and application context objects are created.

Spring also provides some default Factory post processors. For Ex:

**PropertyPlaceholderConfigurer**

This will read the values from a property files and puts the values in our bean property before the application context object is created

<bean calss=”org.springframework.beans.factory.config.**PropertyPlaceholderConfigurer**”>

<property **name=”locations” value=” myProperty.property”**/>

</>

<bean id=”myParentBean” class=”..MyBaseClass” >

<property name=”a” value=”**${userName}**”/>

</>

Here the username is the key present in the myProperty.property file.

**Coding by Interface**

Public Interface myInterface {

Public void callMe();

}

Public class myClass implements myInterface {

Public void callMe(){

….

}

}

<bean id=”myClass” calss=”..Myclass”/>

myInterfaceobj =context.getBean(“myClass”);

obj.callMe();

**@Required**

To validate all the required dependencies are met

Public Class myClass{

Private String myString;

**@Required**

Public void setMyString(String myString){

this.myString = myString;

}

}

<bean class=”org.springframework.beans.factory.annotation.**RequiredAnnotationBeanPoetProcessor**”/>

This class will check for the @Required it will throw exception (at the time of creating the application context object)

**@Autowired**

Public class MyParent {

Private MyChild**child**;

**@Autowired**

Public void setChild (String **child**){

}

}

Here spring will look for a been mapping with **type** first so

<bean id=”myParent” calss=”..MyParent”>

</>

<bean id=”myChild” calss=”..MyChild”>

</>

In the above mapping to auto wire will work.

Now

<bean id=”myParent” calss=”..MyParent”>

</>

<bean id=”**myChild\_1**” calss=”..MyChild”>

</>

<bean id=”**myChild\_2**” calss=”..MyChild”>

</>

In the above mapping it will throw exception because it is having more than one mapping of type MyChild class. So what spring will looks for next is **byname**.

So

<bean id=”myParent” calss=”..MyParent”>

</>

<bean id=”**child**” calss=”..MyChild”>

</>

<bean id=”**myChild\_2**” calss=”..MyChild”>

</>

It will work now cas the id child is same as child in java

Now if we don’t want to change the id name as some other beans could be referring with the old name, then we can use @Qualifier

**@Qualifier**

Public class MyParent {

Private MyChildchild;

@Autowired

**@Qualifier(“myChild\_123”)**

Public void setChild (String child){

}

}

<bean id=”myParent” calss=”..MyParent”>

</>

<bean id=”myChild\_1” calss=”..MyChild”>

**<qualifier name=” myChild\_123”/>**

</>

<bean id=”**myChild\_2**” calss=”..MyChild”>

</>

**JSR – 250 Annotations**

**@Resource**

Is used to achieve DI

Public class MyParent {

Private MyChild**child**;

**@Resource**

Public void setChild (String child){

}

}

<bean id=”myParent” calss=”..MyParent”>

</>

<bean id=”**child**” calss=”..MyChild”>

</>

This will do a DI based on byname. If we have the bean mapping with different name then we need to use **@Resource(name=”myChild\_1”)**

Public class MyParent {

Private MyChild**child**;

**@Resource(name=”myChild\_1”)**

Public void setChild (String child){

}

}

<bean id=”myParent” calss=”..MyParent”>

</>

<bean id=”**myChild\_1**” calss=”..MyChild”>

</>

**@PostConstruct**

Gets executed after the bean is initialized

**@PreDestroy**

Get executed the before the bean is destroyed

Public class MyClass {

**@PostConstruct**

Public void myInit(){

}

**@PreDestroy**

Public void myDestroy(){

}

}

**Using annotation only and no XML**

@Component // for class in standalone application

Public class MyClass{

}

Is equal to <bean id=”myClass” class=”..MyClass”/>

If we want to do the same in Enterprise application i.e. MVC then we need to add stereotype annotations. Those are:

@Controller // Controller

@Service // Service

@Repository // DAO

And to link each other layer or class we will use @Autowire annotation

In XML we need to add only

<context:component-scan base-package=”org.sujay….”/>

**MessageSource to read text from the property file (Available Only in Application Context)**

First create a .property file myProperty.property

Define the message source bean in the applicationcontext.xml

<bean id=”myMessageSource” class=”org.springframework.context.support.ResourceBundleMessageSource”>

<property name=”basenames”>

<list>

<value>myProperty</>

</>

</>

</>

In Java

Public static void main(String[] ad){

ApplicationContext context = new ClassPathXmlApplicationContext(“applicationcontext.xml”);

context.getMessage(“key”);

}

If we want to use this in our bean class then (J2EE Applications):

<bean id=”myClass” class=”..MyClass”>

<property messageSource=”myMessageSource”/>

</>

Public class MyClass{

Private MessageSourcemessageSource; //Spring Interface

Public void SetMessageSource(MessageSourcemessageSource){

this.messageSource = messageSource;

}

Public void printMyMessage(){

This.messageSource.getMessage(“key”);

}

}

Note there are different impl of getMessage like one with a key only one with a key, default value and so on.

**Event Handling in Spring(Available Only in Application Context)**

Event handling promotes loose coupling.

Generally we have three components in Event handling:

1. Event Publisher
2. Event Listener
3. Event

Spring provides below mentioned to handle the event handling:

1. ApplicationListener // interfaces
2. ApplicationEventPublisher // interfaces
3. ApplicationEvent // class

**//Class in which we publish event**

Public class myService**implements ApplicationEventPublisherAware // this will give the publisher obj**{

Private **ApplicationEventPublisher publisher;**

Public void setApplicationEventPublisher(**ApplicationEventPublisher publisher**){

this.publisher = publisher;

}

Public void **sendEmailI**fValidUser(){ // a service method

If(validUser()){

EmailEventemailEvent = new EmailEvent(**this**);

p**ublisher.publishEvent(emailEvent);**

}

}

}

//The Event class

Public class EmailEvent extend **ApplicationEvent**{

**//Need to call the ApplicationEvent constructor**

**publicEmailEvent(Object source){**

**Super(source);**

**}**

Public voidsendEmail(){

Code to send mail

}

}

**//Class which listens to the event**

Public class MyEventLisetner**implements ApplicationListner**{

Public void **onApplicationEvent(ApplicationEvent event){**

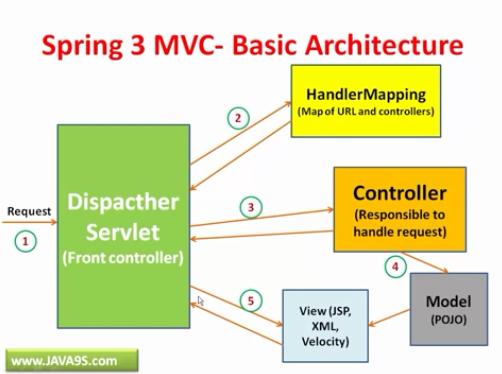
**event.sendEmail();** //Call the sendEmail method of the myEvent class

**}**

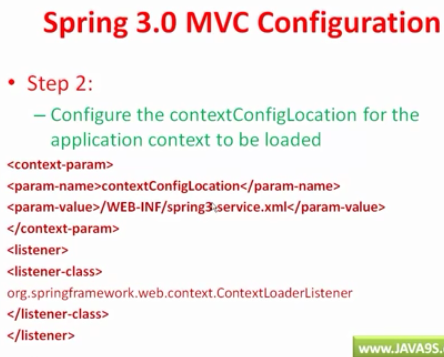
}

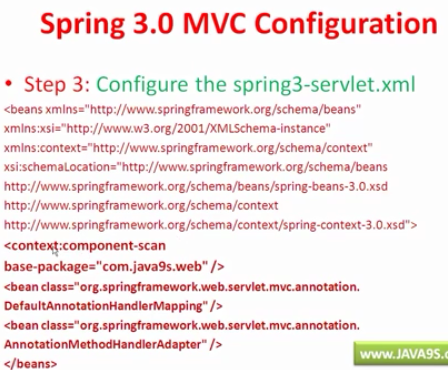
**AOP**

**Spring MVC**

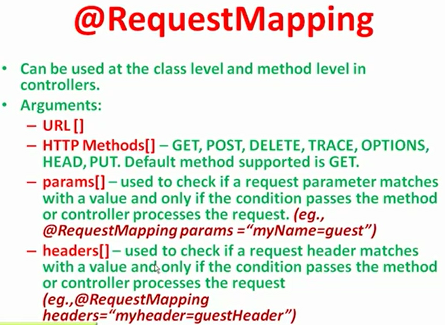


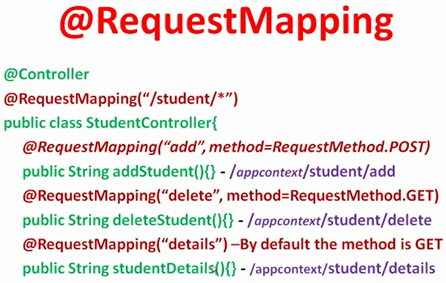


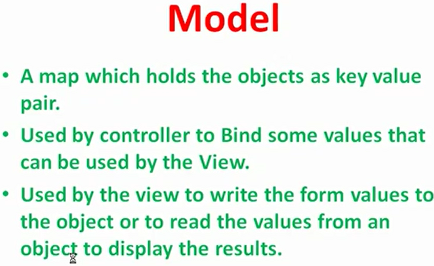




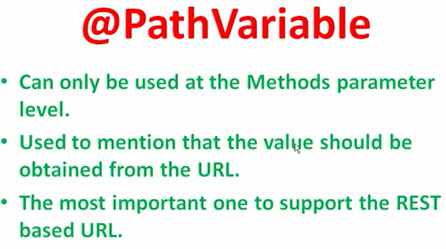
Need to specify the base-packege to let spring know the controller class location and to look for @Controller

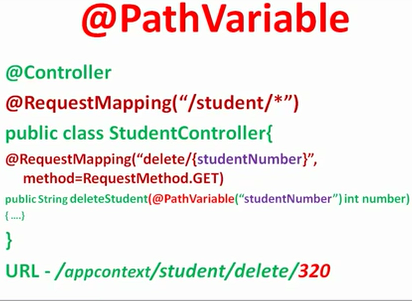




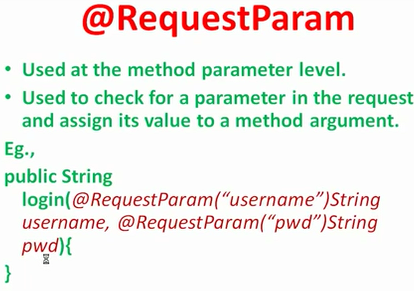


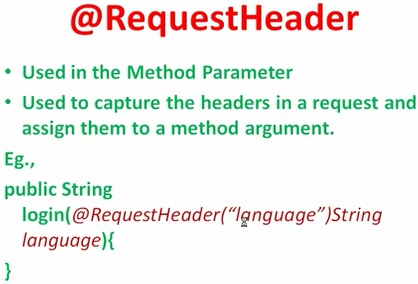


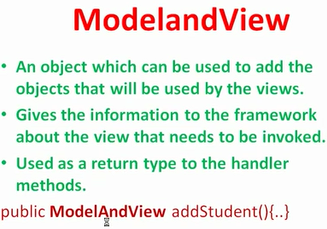


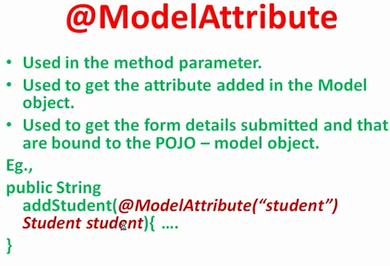


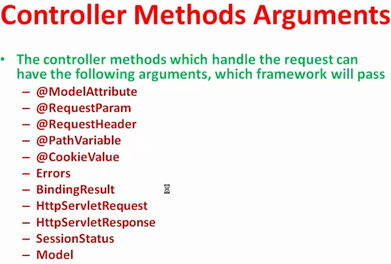
The above url is a RESTFUL url where is No query string concept.

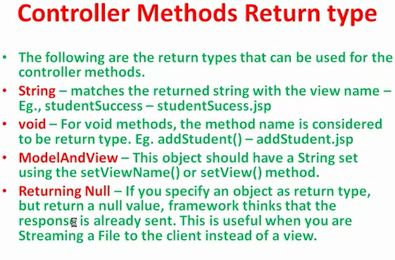












Spring MVC CRUD

1. **public** **class** **EmployeeBean** {
2. **private** Integer id;
3. **private** String name;
4. **private** Integer age;
5. **private** Long salary;
6. **private** String address;
7. Get() set()…
8. @Entity
9. @Table(name="Employee")
10. **public** **class** **Employee** **implements** Serializable{
12. **private** **static** **final** **long** serialVersionUID = -723583058586873479L;
14. @Id
15. @GeneratedValue(strategy=GenerationType.AUTO)
16. @Column(name = "empid")
17. **private** Integer empId;
19. @Column(name="empname")
20. **private** String empName;
22. @Column(name="empaddress")
23. **private** String empAddress;
25. @Column(name="salary")
26. **private** Long salary;
28. @Column(name="empAge")
29. **private** Integer empAge;
30. Get() set()….
31. @Repository("employeeDao")
32. **public** **class** **EmployeeDaoImpl** **implements** EmployeeDao {
34. @Autowired
35. **private** SessionFactory sessionFactory;
37. **public** **void** addEmployee(Employee employee) {
38. sessionFactory.getCurrentSession().saveOrUpdate(employee);
39. }
41. @SuppressWarnings("unchecked")
42. **public** List<Employee> listEmployeess() {
43. **return** (List<Employee>) sessionFactory.getCurrentSession().createCriteria(Employee.**class**).list();
44. }
46. **public** Employee getEmployee(**int** empid) {
47. **return** (Employee) sessionFactory.getCurrentSession().get(Employee.**class**, empid);
48. }
50. **public** **void** deleteEmployee(Employee employee) {
51. sessionFactory.getCurrentSession().createQuery("DELETE FROM Employee WHERE empid = "+employee.getEmpId()).executeUpdate();
52. }
53. **@Service**("employeeService")
54. **@Transactional**(propagation = Propagation.SUPPORTS, readOnly = **true**)
55. **public** **class** **EmployeeServiceImpl** **implements** EmployeeService {
57. @Autowired
58. **private** EmployeeDao employeeDao;
60. @Transactional(propagation = Propagation.REQUIRED, readOnly = **false**)
61. **public** **void** addEmployee(Employee employee) {
62. employeeDao.addEmployee(employee);
63. }
65. **public** List<Employee> listEmployeess() {
66. **return** employeeDao.listEmployeess();
67. }
69. **public** Employee getEmployee(**int** empid) {
70. **return** employeeDao.getEmployee(empid);
71. }
73. **public** **void** deleteEmployee(Employee employee) {
74. employeeDao.deleteEmployee(employee);
75. }
76. **@Controller**
77. **public** **class** **EmployeeController** {
79. @Autowired
80. **private** EmployeeService employeeService;
82. @RequestMapping(value = "/save", method = RequestMethod.POST)
83. **public** ModelAndView saveEmployee(@ModelAttribute("command")EmployeeBean employeeBean,
84. BindingResult result) {
85. Employee employee = prepareModel(employeeBean);
86. employeeService.addEmployee(employee);
87. **return** **new** ModelAndView("redirect:/add.html");
88. }
90. @RequestMapping(value="/employees", method = RequestMethod.GET)
91. **public** ModelAndView listEmployees() {
92. Map<String Object> model = **new** HashMap<String Object>();
93. model.put("employees",  prepareListofBean(employeeService.listEmployeess()));
94. **return** **new** ModelAndView("employeesList", model);
95. }
97. @RequestMapping(value = "/add", method = RequestMethod.GET)//**1st method**
98. **public** ModelAndView addEmployee(@ModelAttribute("command")EmployeeBean employeeBean,
99. BindingResult result) {
100. Map<String, Object> model = **new** HashMap<String, Object>();
101. model.put("employees",  prepareListofBean(employeeService.listEmployeess()));
102. **return** **new** ModelAndView("addEmployee", model);
103. }
105. @RequestMapping(value = "/index", method = RequestMethod.GET)
106. **public** ModelAndView welcome() {
107. **return** **new** ModelAndView("index");
108. }
110. @RequestMapping(value = "/delete", method = RequestMethod.GET)
111. **public** ModelAndView editEmployee(@ModelAttribute("command")EmployeeBean employeeBean,
112. BindingResult result) {
113. employeeService.deleteEmployee(prepareModel(employeeBean));
114. Map<String<b>,</b> Object> model = **new** HashMap<String, Object>();
115. model.put("employee", **null**);
116. model.put("employees",  prepareListofBean(employeeService.listEmployeess()));
117. **return** **new** ModelAndView("addEmployee", model);
118. }
120. @RequestMapping(value = "/edit", method = RequestMethod.GET)
121. **public** ModelAndView deleteEmployee(@ModelAttribute("command")EmployeeBean employeeBean,
122. BindingResult result) {
123. Map<String, Object> model = **new** HashMap<String, Object>();
124. model.put("employee", prepareEmployeeBean(employeeService.getEmployee(employeeBean.getId())));
125. model.put("employees",  prepareListofBean(employeeService.listEmployeess()));
126. **return** **new** ModelAndView("addEmployee", model);
127. }
129. **private** Employee prepareModel(EmployeeBean employeeBean){
130. Employee employee = **new** Employee();
131. employee.setEmpAddress(employeeBean.getAddress());
132. employee.setEmpAge(employeeBean.getAge());
133. employee.setEmpName(employeeBean.getName());
134. employee.setSalary(employeeBean.getSalary());
135. employee.setEmpId(employeeBean.getId());
136. employeeBean.setId(**null**);
137. **return** employee;
138. }
140. **private** List<EmployeeBean> prepareListofBean(List<Employee> employees){
141. List<employeebean> beans = **null**;
142. **if**(employees != **null** && !employees.isEmpty()){
143. beans = **new** ArrayList<EmployeeBean>();
144. EmployeeBean bean = **null**;
145. **for**(Employee employee : employees){
146. bean = **new** EmployeeBean();
147. bean.setName(employee.getEmpName());
148. bean.setId(employee.getEmpId());
149. bean.setAddress(employee.getEmpAddress());
150. bean.setSalary(employee.getSalary());
151. bean.setAge(employee.getEmpAge());
152. beans.add(bean);
153. }
154. }
155. **return** beans;
156. }
158. **private** EmployeeBean prepareEmployeeBean(Employee employee){
159. EmployeeBean bean = **new** EmployeeBean();
160. bean.setAddress(employee.getEmpAddress());
161. bean.setAge(employee.getEmpAge());
162. bean.setName(employee.getEmpName());
163. bean.setSalary(employee.getSalary());
164. bean.setId(employee.getEmpId());
165. **return** bean;
166. }

***addEmployee.jsp***

1. **<**%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%**>**
2. **<**%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%**>**
3. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"**>**
4. **<html>**
5. **<head>**
6. **<meta** http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"**>**
7. **<title>**Spring MVC Form Handling**</title>**
8. **</head>**
9. **<body>**
10. **<h2>**Add Employee Data**</h2>**
11. **<form:form** method="POST" action="/sdnext/save.html"**>**
12. **<table>**
13. **<tr>**
14. **<td><form:label** path="id"**>**Employee ID:**</form:label></td>**
15. **<td><form:input** path="id" value="${employee.id}" readonly="true"**/></td>**
16. **</tr>**
17. **<tr>**
18. **<td><form:label** path="name"**>**Employee Name:**</form:label></td>**
19. **<td><form:input** path="name" value="${employee.name}"**/></td>**
20. **</tr>**
21. **<tr>**
22. **<td><form:label** path="age"**>**Employee Age:**</form:label></td>**
23. **<td><form:input** path="age" value="${employee.age}"**/></td>**
24. **</tr>**
25. **<tr>**
26. **<td><form:label** path="salary"**>**Employee Salary:**</form:label></td>**
27. **<td><form:input** path="salary" value="${employee.salary}"**/></td>**
28. **</tr>**
30. **<tr>**
31. **<td><form:label** path="address"**>**Employee Address:**</form:label></td>**
32. **<td><form:input** path="address" value="${employee.address}"**/></td>**
33. **</tr>**
34. **<tr>**
35. **<td** colspan="2"**><input** type="submit" value="Submit"**/></td>**
36. **</tr>**
37. **</table>**
38. **</form:form>**
40. **<c:if** test="${!empty employees}"**>**
41. **<h2>**List Employees**</h2>**
42. **<table** align="left" border="1"**>**
43. **<tr>**
44. **<th>**Employee ID**</th>**
45. **<th>**Employee Name**</th>**
46. **<th>**Employee Age**</th>**
47. **<th>**Employee Salary**</th>**
48. **<th>**Employee Address**</th>**
49. **<th>**Actions on Row**</th>**
50. **</tr>**
52. **<c:forEach** items="${employees}" var="employee"**>**
53. **<tr>**
54. **<td><c:out** value="${employee.id}"**/></td>**
55. **<td><c:out** value="${employee.name}"**/></td>**
56. **<td><c:out** value="${employee.age}"**/></td>**
57. **<td><c:out** value="${employee.salary}"**/></td>**
58. **<td><c:out** value="${employee.address}"**/></td>**
59. **<td** align="center"**><a** href="edit.html?id=${employee.id}"**>**Edit**</a>** | **<a** href="delete.html?id=${employee.id}"**>**Delete**</a></td>**
60. **</tr>**
61. **</c:forEach>**
62. **</table>**
63. **</c:if>**
64. **</body>**
65. **</html>**

***employeesList.jsp***

1. **<**%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%**>**
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"**>**
3. **<html>**
4. **<head>**
5. **<title>**All Employees**</title>**
6. **</head>**
7. **<body>**
8. **<h1>**List Employees**</h1>**
9. **<h3><a** href="add.html"**>**Add More Employee**</a></h3>**
11. **<c:if** test="${!empty employees}"**>**
12. **<table** align="left" border="1"**>**
13. **<tr>**
14. **<th>**Employee ID**</th>**
15. **<th>**Employee Name**</th>**
16. **<th>**Employee Age**</th>**
17. **<th>**Employee Salary**</th>**
18. **<th>**Employee Address**</th>**
19. **</tr>**
21. **<c:forEach** items="${employees}" var="employee"**>**
22. **<tr>**
23. **<td><c:out** value="${employee.id}"**/></td>**
24. **<td><c:out** value="${employee.name}"**/></td>**
25. **<td><c:out** value="${employee.age}"**/></td>**
26. **<td><c:out** value="${employee.salary}"**/></td>**
27. **<td><c:out** value="${employee.address}"**/></td>**
28. **</tr>**
29. **</c:forEach>**
30. **</table>**
31. **</c:if>**
32. **</body>**
33. **</html>**

Index.jsp

1. **<**%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2. pageEncoding="ISO-8859-1"%**>**
3. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"**>**
4. **<html>**
5. **<head>**
6. **<meta** http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"**>**
7. **<title>**Spring3MVC with Hibernate3 CRUD Example using Annotations**</title>**
8. **</head>**
9. **<body>**
10. **<h2>**Spring3MVC with Hibernate3 CRUD Example using Annotations**</h2>**
11. **<h2>**1. **<a** href="employees.html"**>**List of Employees**</a></h2>**
12. **<h2>**2. **<a** href="add.html"**>**Add Employee**</a></h2>**
13. **</body>**
14. **</html>**

Web.xml

1. **<servlet>**
2. **<servlet-name>**springCrud**</servlet-name>**
3. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
4. **<init-param>**
5. **<param-name>**contextConfigLocation**</param-name><param-value>**/WEB-INF/config/sdnext-servlet.xml**</param-value></init-param>**
6. **<load-on-startup>**1**</load-on-startup>**
7. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**sdnext**</servlet-name>**
11. **<url-pattern>**\*.html**</url-pattern>**
12. **</servlet-mapping>**
14. **<welcome-file-list>**
15. **<welcome-file>**index.html**</welcome-file>**
16. **</welcome-file-list>**
18. **</web-app>**

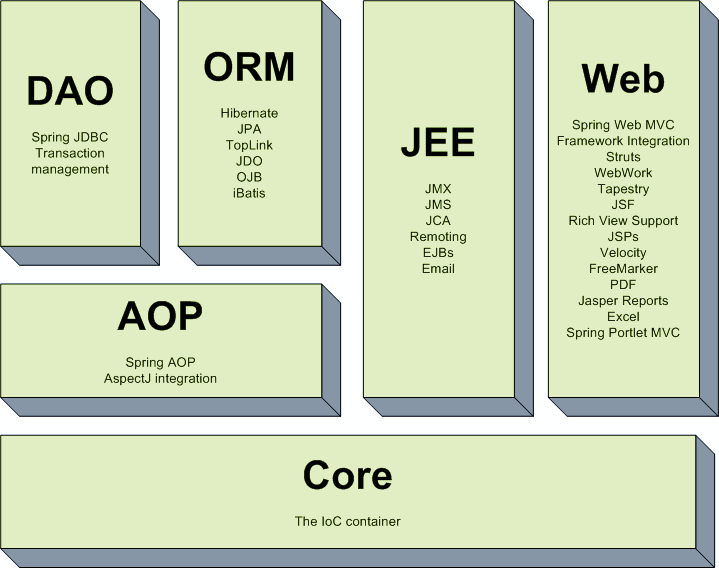
**springCrud-servlet.xml**

1. **<context:property-placeholder** location="classpath:resources/database.properties"**>**
2. **</context:property-placeholder>**
3. **<context:component-scan** base-package="com.dineshonjava"**>**
4. **</context:component-scan>**
6. **<tx:annotation-driven** transaction-manager="hibernateTransactionManager"**>**
7. **</tx:annotation-driven>**
9. **<bean** class="org.springframework.web.servlet.view.InternalResourceViewResolver" id="jspViewResolver"**>**
10. **<property** name="viewClass" value="org.springframework.web.servlet.view.JstlView"**></property>**
11. **<property** name="prefix" value="/WEB-INF/views/"**></property>**
12. **<property** name="suffix" value=".jsp"**></property>**
13. **</bean>**
15. **<bean** class="org.springframework.jdbc.datasource.DriverManagerDataSource" id="dataSource"**>**
16. **<property** name="driverClassName" value="${database.driver}"**></property>**
17. **<property** name="url" value="${database.url}"**></property>**
18. **<property** name="username" value="${database.user}"**></property>**
19. **<property** name="password" value="${database.password}"**></property>**
20. **</bean>**
22. **<bean** class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean" id="sessionFactory"**>**
23. **<property** name="dataSource" ref="dataSource"**></property>**
24. **<property** name="annotatedClasses"**>**
25. **<list>**
26. **<value>**com.dineshonjava.model.Employee**</value>**
27. **</list>**
28. **</property>**
29. **<property** name="hibernateProperties"**>**
30. **<props>**
31. **<prop** key="hibernate.dialect"**>**${hibernate.dialect}**</prop>**
32. **<prop** key="hibernate.show\_sql"**>**${hibernate.show\_sql}**</prop>**
33. **<prop** key="hibernate.hbm2ddl.auto"**>**${hibernate.hbm2ddl.auto}  **</prop>**
34. **</props>**
35. **</property>**
36. **</bean>**
38. **<bean** class="org.springframework.orm.hibernate3.HibernateTransactionManager" id="hibernateTransactionManager"**>**
39. **<property** name="sessionFactory" ref="sessionFactory"**></property>**
40. **</bean>**

**</beans>**

What are the different modules in Spring framework?

**What is the structure of Spring framework?**

[](http://cdn.javabeat.net/wp-content/uploads/2009/02/spring-framework.png)

**5) What is the Core container module?**

This module is provides the fundamental functionality of the spring framework. In this module BeanFactory is the heart of any spring-based application. The entire framework was built on the top of this module. This module makes the Spring container.

**6) What is Application context module?**

The Application context module makes spring a framework. This module extends the concept of BeanFactory, providing support for internationalization (I18N) messages, application lifecycle events, and validation. This module also supplies many enterprise services such JNDI access, EJB integration, remoting, and scheduling. It also provides support to other framework.

**7) What is AOP module?**

The AOP module is used for developing aspects for our Spring-enabled application. Much of the support has been provided by the AOP Alliance in order to ensure the interoperability between Spring and other AOP frameworks. This module also introduces metadata programming to Spring. Using Spring’s metadata support, we will be able to add annotations to our source code that instruct Spring on where and how to apply aspects.

**8)What is JDBC abstraction and DAO module?**

Using this module we can keep up the database code clean and simple, and prevent problems that result from a failure to close database resources. A new layer of meaningful exceptions on top of the error messages given by several database servers is bought in this module. In addition, this module uses Spring’s AOP module to provide transaction management services for objects in a Spring application.

**12) What is a BeanFactory?**

A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application’s configuration and dependencies from the actual application code.

**14) What is Spring configuration file?**

Spring configuration file is an XML file. This file contains the classes information and describes how these classes are configured and introduced to each other.

**16) What is XMLBeanFactory?**

BeanFactory has many implementations in Spring. But one of the most useful one is org.springframework.beans.factory.xml.XmlBeanFactory, which loads its beans based on the definitions contained in an XML file. To create an XmlBeanFactory, pass a java.io.InputStream to the constructor. The InputStream will provide the XML to the factory. For example, the following code snippet uses a java.io.FileInputStream to provide a bean definition XML file to XmlBeanFactory.

|  |  |  |
| --- | --- | --- |
|  | BeanFactory factory = new XmlBeanFactory( | |
|  | new FileInputStream('beans.xml')); |

To retrieve the bean from a BeanFactory, call the getBean() method by passing the name of the bean you want to retrieve.

|  |  |
| --- | --- |
|  | MyBeanmyBean = (MyBean) factory.getBean('myBean'); |

**17) What are important ApplicationContext implementations in spring framework?**

**ClassPathXmlApplicationContext** – This context loads a context definition from an XML file located in the class path, treating context definition files as class path resources.

**FileSystemXmlApplicationContext** – This context loads a context definition from an XML file in the filesystem.

**XmlWebApplicationContext** – This context loads the context definitions from an XML file contained within a web application.

**18) Explain Bean lifecycle in Spring framework?**

The spring container finds the bean’s definition from the XML file and instantiates the bean.

Using the dependency injection, spring populates all of the properties as specified in the bean definition.

If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean’s ID.

If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.

If there are any BeanPostProcessors associated with the bean, their post- ProcessBeforeInitialization() methods will be called.

If an init-method is specified for the bean, it will be called.

Finally, if there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.

**21) What are singleton beans and how can you create prototype beans?**

Beans defined in spring framework are singleton beans. There is an attribute in bean tag named ‘singleton’ if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default it is set to true. So, all the beans in spring framework are by default singleton beans.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | <beans> | | |
| 2 | <bean id='bar' class='com.act.Foo' | | | |
| 3 | singleton='false'/> | |
| 4 | </beans> |

**22) What are the important beans lifecycle methods?**

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.

**23) How can you override beans default lifecycle methods?**

The bean tag has two more important attributes with which you can define your own custom initialization and destroy methods. Here I have shown a small demonstration. Two new methods fooSetup and fooTeardown are to be added to your Foo class.

|  |  |
| --- | --- |
| 1 | <beans> |
| 2 | <bean id='bar' class='com.act.Foo' | |
| 3 | init-method='fooSetup' destroy='fooTeardown'/> | | | |
| 4 | </beans> | | |

**24) What are Inner Beans?**

When wiring beans, if a bean element is embedded to a property tag directly, then that bean is said to the Inner Bean. The drawback of this bean is that it cannot be reused anywhere else.

**26) What is Auto wiring?**

You can wire the beans as you wish. But spring framework also does this work for you. It can auto wire the related beans together. All you have to do is just set the autowire attribute of bean tag to an autowire type.

|  |  |  |
| --- | --- | --- |
| 1 | <beans> | |
| 2 | <bean id='bar' class='com.act.Foo' Autowire='autowire type'/> | | |
| 3 | </beans> |

**27) What are different types of Autowire types?**

There are four different types by which autowiring can be done.

byName

byType

constructor

autodetect

**28) What are the different types of events related to Listeners?**

There are a lot of events related to ApplicationContext of spring framework. All the events are subclasses of org.springframework.context.Application-Event. They are

ContextClosedEvent – This is fired when the context is closed.

ContextRefreshedEvent – This is fired when the context is initialized or refreshed.

RequestHandledEvent – This is fired when the web context handles any request.

**29) What is an Aspect?**

An aspect is the cross-cutting functionality that you are implementing. It is the aspect of your application you are modularizing. An example of an aspect is logging. Logging is something that is required throughout an application. However, because applications tend to be broken down into layers based on functionality, reusing a logging module through inheritance does not make sense. However, you can create a logging aspect and apply it throughout your application using AOP.

**30) What is a Jointpoint?**

A joinpoint is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modified. These are the points where your aspect’s code can be inserted into the normal flow of your application to add new behavior.

**31) What is an Advice?**

Advice is the implementation of an aspect. It is something like telling your application of a new behavior. Generally, and advice is inserted into an application at joinpoints.

**32) What is a Pointcut?**

A pointcut is something that defines at what joinpoints an advice should be applied. Advices can be applied at any joinpoint that is supported by the AOP framework. These Pointcuts allow you to specify where the advice can be applied.

**33) What is an Introduction in AOP?**

An introduction allows the user to add new methods or attributes to an existing class. This can then be introduced to an existing class without having to change the structure of the class, but give them the new behavior and state.

34) What is a Target?

A target is the class that is being advised. The class can be a third party class or your own class to which you want to add your own custom behavior. By using the concepts of AOP, the target class is free to center on its major concern, unaware to any advice that is being applied.

35) What is a Proxy?

A proxy is an object that is created after applying advice to a target object. When you think of client objects the target object and the proxy object are the same.

36) What is meant by Weaving?

The process of applying aspects to a target object to create a new proxy object is called as Weaving. The aspects are woven into the target object at the specified joinpoints.

38) What are the different advice types in spring?

* + **Around :** Intercepts the calls to the target method
  + **Before :** This is called before the target method is invoked
  + **After :** This is called after the target method is returned
  + **Throws :** This is called when the target method throws and exception
* Around : org.aopalliance.intercept.MethodInterceptor
* Before : org.springframework.aop.BeforeAdvice
* After : org.springframework.aop.AfterReturningAdvice
* Throws : org.springframework.aop.ThrowsAdvice

43) How can you configure a bean to get DataSource from JNDI?

|  |  |
| --- | --- |
| 1 | <beanid='dataSource' |
| 2 | class='org.springframework.jndi.JndiObjectFactoryBean'> | |

|  |  |
| --- | --- |
| 3 | <propertyname='jndiName'> |
| 4 | <value>java:comp/env/jdbc/myDatasource</value> | |

|  |  |  |
| --- | --- | --- |
| 5 | </property> | |
| 6 | </bean> |

44) How can you create a DataSource connection pool?

|  |  |
| --- | --- |
| 1 | <beanid='dataSource' |
| 2 | class='org.apache.commons.dbcp.BasicDataSource'> | |

|  |  |
| --- | --- |
| 3 | <propertyname='driver'> |
| 4 | <value>${db.driver}</value> | |

|  |  |
| --- | --- |
| 5 | </property> |
| 6 | <propertyname='url'> | |

|  |  |  |
| --- | --- | --- |
| 7 | <value>${db.url}</value> | |
| 8 | </property> |

|  |  |  |
| --- | --- | --- |
| 9 | <propertyname='username'> | |
| 10 | | <value>${db.username}</value> | |

|  |  |
| --- | --- |
| 11 | </property> |
| 12 | <propertyname='password'> | |

|  |  |  |
| --- | --- | --- |
| 13 | <value>${db.password}</value> | |
| 14 | </property> |

|  |  |
| --- | --- |
| 15 | </bean> |
|  |  |

45) How JDBC can be used more efficiently in spring framework?

JDBC can be used more efficiently with the help of a template class provided by spring framework called as **JdbcTemplate**.

46) How JdbcTemplate can be used?

With use of Spring JDBC framework the burden of resource management and error handling is reduced a lot. So it leaves developers to write the statements and queries to get the data to and from the database.

|  |  |
| --- | --- |
| 1 | <strong>JdbcTemplate</strong> template = new <strong>JdbcTemplate</strong>(myDataSource); |

A simple DAO class looks like this.

|  |  |  |
| --- | --- | --- |
| 1 | publicclassStudentDaoJdbc implementsStudentDao { | |
| 2 | privateJdbcTemplatejdbcTemplate; |

|  |  |  |
| --- | --- | --- |
| 3 | publicvoidsetJdbcTemplate(JdbcTemplatejdbcTemplate) { | |
| 4 | this.jdbcTemplate = jdbcTemplate; |

|  |  |  |
| --- | --- | --- |
| 5 | } more.. | |
| 6 | } |

The configuration is shown below.

|  |  |  |
| --- | --- | --- |
| 1 | <beanid='jdbcTemplate'class='org.springframework.jdbc.core.JdbcTemplate'> | |
| 2 | <propertyname='dataSource'> |

|  |  |  |
| --- | --- | --- |
| 3 | <refbean='dataSource'/> | |
| 4 | </property> |

|  |  |
| --- | --- |
| 5 | </bean> |
| 6 | <beanid='studentDao'class='StudentDaoJdbc'> | |

|  |  |  |
| --- | --- | --- |
| 7 | <propertyname='jdbcTemplate'> | |
| 8 | <refbean='jdbcTemplate'/> |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | </property> | | |
| 10 | | </bean> |

|  |  |  |
| --- | --- | --- |
| 11 | <beanid='courseDao'class='CourseDaoJdbc'> | |
| 12 | <propertyname='jdbcTemplate'> |

|  |  |  |
| --- | --- | --- |
| 13 | <refbean='jdbcTemplate'/> | |
| 14 | </property> |

|  |  |
| --- | --- |
| 15 | </bean> |

47) How do you write data to backend in spring using JdbcTemplate?

The JdbcTemplate uses several of these callbacks when writing data to the database. The usefulness you will find in each of these interfaces will vary. There are two simple interfaces. One is **PreparedStatementCreator** and the other interface is **BatchPreparedStatementSetter**.