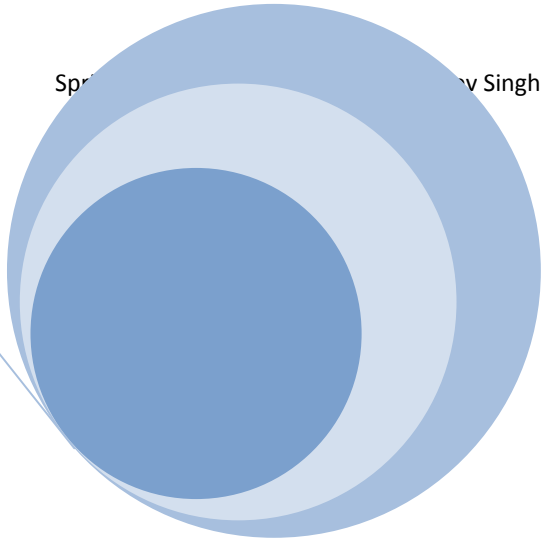
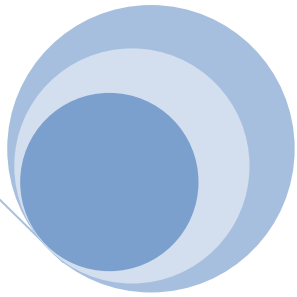
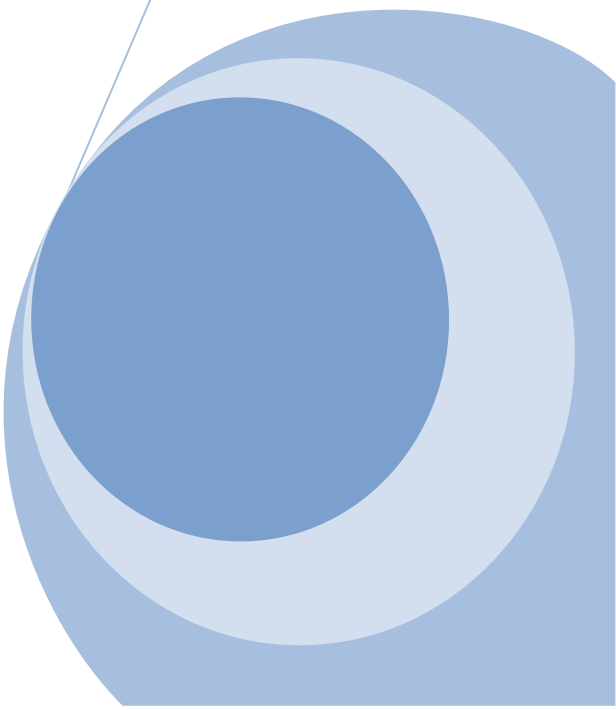






Spring Framework Interview Questions

Version 1.0

Sanjeev Kumar Singh
6/2/2011

Spring –Core Questions

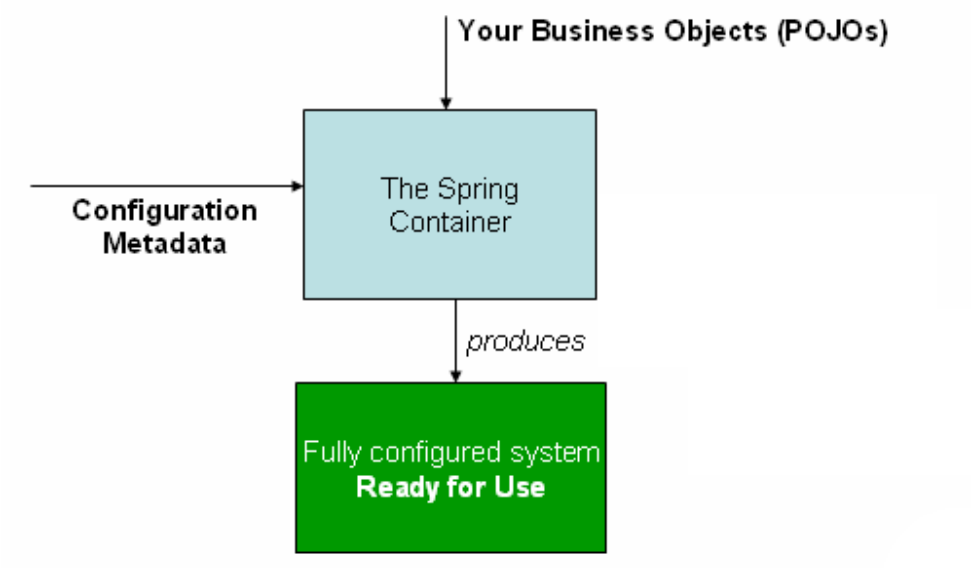
Version 1.0

Sanjeev Kumar Singh

1. What is Spring?

Spring is a lightweight inversion of control and aspect-oriented container framework which managed Business Objects and their relationship. Spring is modular and has been divided logically into independent packages, which can function independently.

The architects of an application have the flexibility to implement just a few spring packages and leave out most of the packages in spring. Whatever Business Components we write in spring are POJO (Plain Old Java Object) or POJI (Plain Old Java Interface) only. POJO/POJI refers to Classes or Interfaces that doesn't specially extend of implement third-party Implementations. The main advantage of having most of the Classes or Interfaces as POJO/POJI in an Application is that they will facilitate easy Unit Testing in the Application.



MyServlet.java

```
class MyServlet extends HttpServlet {

}
```

The problem with the above class definition is that it is not a POJO, because it is extending the HttpServlet class. When this class wants to undergo Unit Testing, someone has to start the Web/Application Server where it is

actually deployed to ascertain the functionality of this class, because of the extension of the HttpServlet class which makes sense in the context of a running Server only. *Since Spring Framework doesn't provide tight coupling between the Business Objects, it is fast to do Unit testing so that TDD (Test Driven Development) can be made easily possible.*

2. What are the benefits of Spring?

Following are the benefits of Spring framework

- Lightweight:

spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 1MB. And the processing overhead is also very negligible.

- Inversion of control (IOC):

Loose coupling is achieved in spring using the technique Inversion of Control. The objects give their dependencies instead of creating or looking for dependent objects.

- Aspect oriented (AOP):

Spring supports Aspect oriented programming and enables cohesive development by separating application business logic from system services.

- Container:

Spring contains and manages the life cycle and configuration of application objects.

- MVC Framework:

Spring comes with MVC web application framework, built on core Spring functionality. This framework is highly configurable via strategy interfaces, and accommodates multiple view technologies like JSP, Velocity, Tiles, iText, and POI. But other frameworks can be easily used instead of Spring MVC Framework.

- Transaction Management:

Spring framework provides a generic abstraction layer for transaction management. This allowing the developer to add the pluggable transaction managers, and making it easy to demarcate transactions without dealing with low-level issues. Spring's transaction support is not tied to J2EE environments and it can be also used in container less environments.

- JDBC Exception Handling:

The JDBC abstraction layer of the Spring offers a meaningful exception hierarchy, which simplifies the error handling strategy. Integration with Hibernate, JDO, and iBATIS: Spring provides best Integration services with Hibernate, JDO and iBATIS

3. What are the different modules in Spring framework?

There are number of modules present in spring framework but 7 modules are most important.

i. Spring AOP

ii. Spring ORM

iii. Spring Web

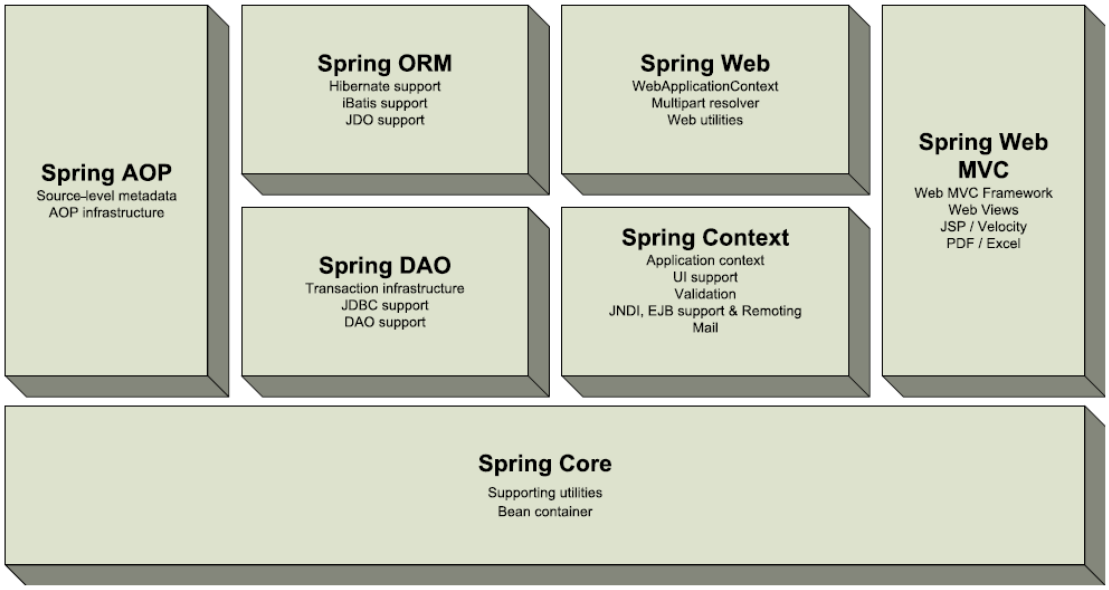
iv. Spring DAO

v. Spring Context

vi. Spring web MVC

vii. Spring Core

4. What is the structure of Spring framework?



Overview of the the Spring Framework

5. What is the Core container module?

Spring core package is used to handle the dependency injection and for creating the BEAN factory. Its primary component is the **'BeanFactory'**, an implementation of the Factory pattern. The BeanFactory applies the IOC pattern to separate an application's configuration and dependency specification from the actual application code. This module makes the spring container.

6. What is Application context module?

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 It also support the text messaging using the resource bundle, event propagation

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?

The DAO package is used to create the JDBC connection which avoids the tedious work of creating connections again and again according to the Vendor specific database. It also used to transaction management. The Spring JDBC DAO abstraction layer offers a meaningful exception hierarchy for managing the exception handling and error messages thrown by different database vendors. The exception hierarchy simplifies error handling and greatly reduces the amount of exception code you need to write, such as opening and closing connections.

9. What are object/relational mapping integration module?

The Spring framework plugs into several ORM frameworks to provide its Object Relational tool, including JDO, Hibernate, and iBatis SQL Maps. All of these comply to Spring's generic transaction and DAO exception hierarchies and with the help of this package we create the POJO mappings with the database tables.

10. What is web module?

Spring comes with a full-featured MVC framework for building web applications. Spring's Web package provides basic web-oriented integration features, such as multipart functionality, initialization of contexts using servlet listeners and a web-oriented application context. Although Spring can easily be integrated with other MVC frameworks, such as Struts, Spring's MVC framework uses IoC to provide for a clean separation of controller

logic from business objects. It also allows you to declaratively bind request parameters to your business objects. It also can take advantage of any of Spring's other services, such as I18N messaging and validation.

11. What is Spring Web MVC?

The MVC framework is a full-featured MVC implementation for building Web applications. The MVC framework is highly configurable via strategy interfaces and accommodates numerous view technologies including JSP, Velocity, Tiles and the generation of PDF and Excel Files.

12. What is a BeanFactory?

The BeanFactory is the actual container which instantiates, configures, and manages a number of beans. These beans typically collaborate with one another, and thus have dependencies between themselves. These dependencies are reflected in the configuration data used by the BeanFactory.

A BeanFactory is represented by the interface **org.springframework.beans.factory.BeanFactory**, for which there are multiple implementations. The most commonly used simple BeanFactory implementation is **org.springframework.beans.factory.xml.XmlBeanFactory**.

```
Resource res = new FileSystemResource("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(res);
```

Or

```
ClassPathResource res = new ClassPathResource("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(res);
```

Or

```
ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(

    new String[] {"applicationContext.xml", "applicationContext-part2.xml"});

// of course, an ApplicationContext is just a BeanFactory

BeanFactory factory = (BeanFactory) appContext;
```

13. What are the attributes which needs to define while creating the Bean?

The bean class

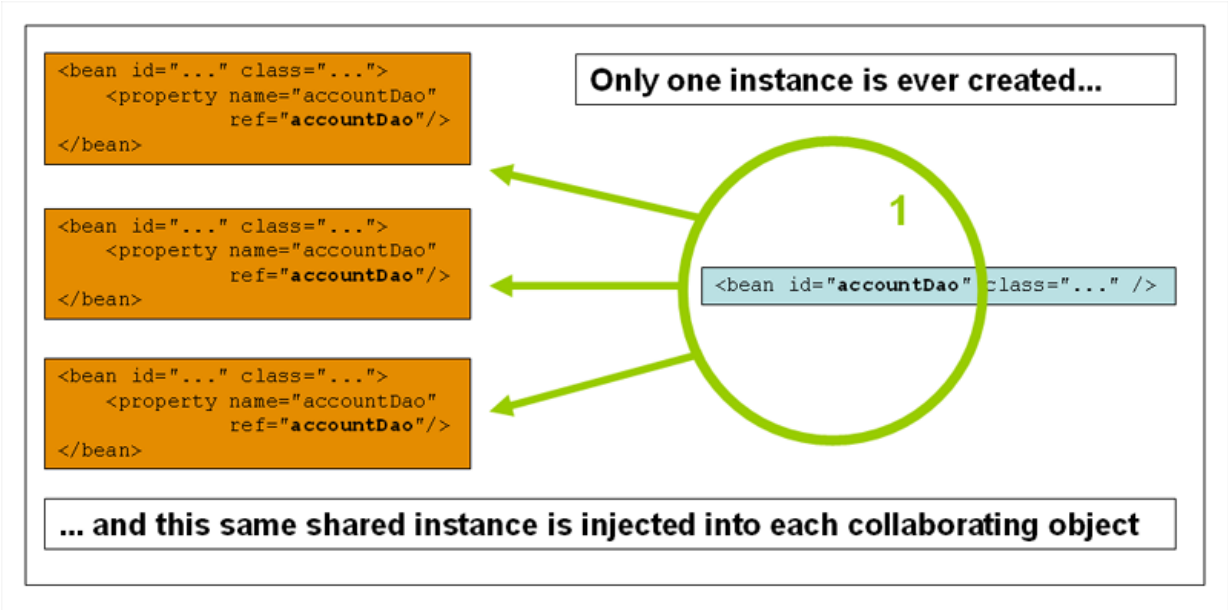
BeanFactory itself directly creates the bean by calling its constructor (equivalent to Java code calling new), the class attribute specifies the class of the bean to be constructed.

The bean identifiers (id and name)

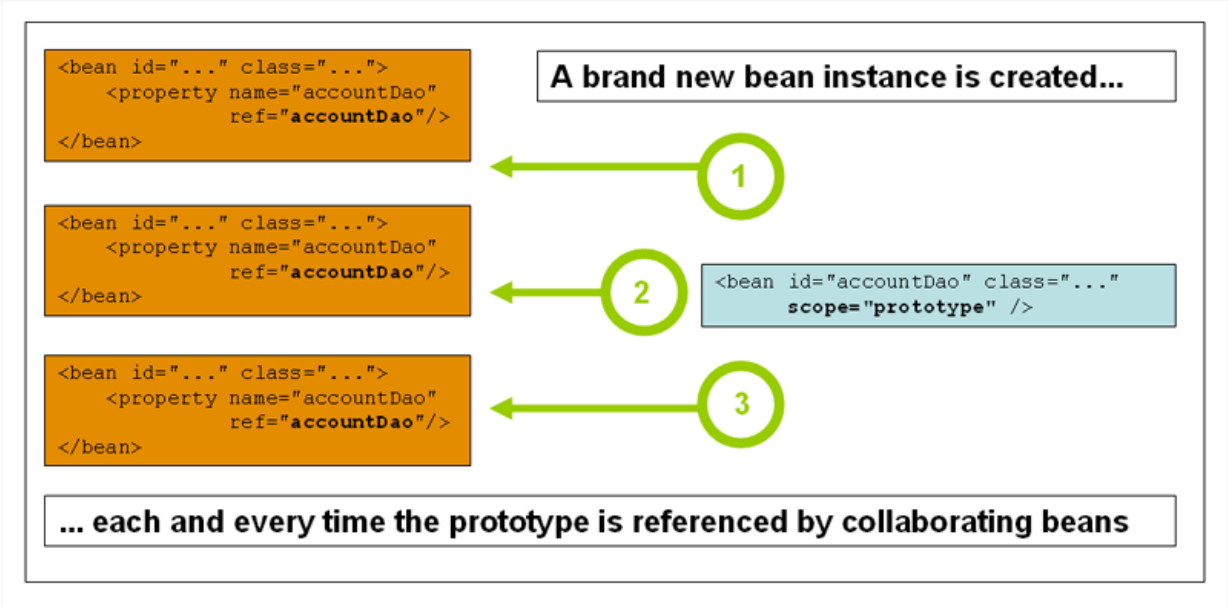
Every bean has one or more ids (also called identifiers, or names; these terms refer to the same thing). These ids must be unique within the BeanFactory or ApplicationContext the bean is hosted in. A bean will almost always have only one id, but if a bean has more than one id, the extra ones can essentially be considered aliases. We may also or instead specify one or more bean ids (separated by a comma (,) or semicolon (;) via the name attribute.

To singleton or not to singleton

Beans are defined to be deployed in one of two modes: singleton or non-singleton. (The latter is also called a prototype, although the term is used loosely as it doesn't quite fit). When a bean is a singleton, only one shared instance of the bean will be managed and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned.



The non-singleton, prototype mode of a bean deployment results in the creation of a new bean instance every time a request for that specific bean is done. This is ideal for situations where for example each user needs an independent user object or something similar.



Beans are deployed in singleton mode by default, unless you specify otherwise.

<bean id="exampleBean" class="examples.ExampleBean" singleton="false"/>

<bean name="yetAnotherExample" class="examples.ExampleBeanTwo" singleton="true"/>

14. What is Bean Aliasing?

In a bean definition itself, you may supply more than one name for the bean, by using a combination of up to one name specified via the id attribute, and any number of other names via the name attribute. All these names can be considered equivalent aliases to the same bean, and are useful for some situations, such as allowing each component used in an application to refer to a common dependency using a bean name that is specific to that component itself.

<alias name="fromName" alias="toName"/>

In this case, a bean in the same container which is named 'fromName', may also after the use of this alias definition, be referred to as 'toName'.

15. How you will create a Bean via constructor?

When creating a bean using the constructor approach, all normal classes are usable by Spring and compatible with Spring. That is, the class being created does not need to implement any specific interfaces or be coded in a specific fashion. Just specifying the bean class should be enough.

```
<bean id="exampleBean"  class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>

<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">

      <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>

      <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>

      <constructor-arg><value>1</value></constructor-arg>

 </bean>
```

16. How you will create a Bean via static factory method?

When defining a bean which is to be created using a static factory method, along with the class attribute which specifies the class containing the static factory method, another attribute named factory-method is needed to specify the name of the factory method itself. Spring expects to be able to call this method

```
<bean id="exampleBean" class="examples.ExampleBean2" factory-method="createInstance"/>
```

17. How you will create a Bean via instance factory method?

To use this mechanism, the class attribute must be left empty, and the factory-bean attribute must specify the name of a bean in the current or an ancestor bean factory which contains the factory method.

```
<!-- The factory bean, which contains a method called    createInstance -->

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

  ...
```

```
</bean>

<!-- The bean to be created via the factory bean -->

<bean id="exampleBean"   factory-bean="myFactoryBean"    factory-method="createInstance"/>
```

18. 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.

15. What does a simple spring application contain?

These applications are like any Java application. They are made up of several classes, each performing a specific purpose within the application. But these classes are configured and introduced to each other through an XML file. This XML file describes how to configure the classes, known as the Spring configuration file.

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.

```
MyBean myBean = (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.

As such bean factory is instantiated by the spring but if we want to instantiate it by the user will do the following way.

```
InputStream is = new FileInputStream("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
```

or

```
ClassPathResource res = new ClassPathResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

or

```
ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml", "applicationContext-part2.xml"});
// of course, an ApplicationContext is just a BeanFactory
BeanFactory factory = (BeanFactory) appContext;
```

18. Explain Bean lifecycle in Spring framework?

1) Instantiate: in this phase container finds the bean's definition and instantiates it.
2) Populate properties: in this phase using the dependency injection all the properties are populated which are specified in the bean definiton (in the xml file).
3) BeanNameAware: If BeanNameAware interface is implemented the factory calls the setBeanName() passing the Bean's Id.
4) If the Bean class implements the BeanClassLoaderAware interface, then the method setBeanClassLoader() method will be called by passing an instance of the ClassLoader object that loaded this bean.
5) BeanFactoryAware: if BeanFactoryAware interface is implemented setBeanFactory() method is called.
6) ApplicationContextAware: This step is valid only if the Application context container is used. In this phase if bean implements the ApplicationcontextAware then setApplicationContext() method is called.

7) Pre-initialization: If any BeanPostProcessors are associated with the bean then their postProcessBeforeInitialization() methods will be called.

8) InitializingBean: If an init-methodis specified for the bean then it is called.

9) Call custom init-method: If there are any BeanPostProcessors are associated then postProcessAfterInitialization() is called.

   Above are the lifecycle phase when a bean becomes ready to use in the container. An existing bean can be removed from the container in two ways:

10) DisposableBean: If bean implements the DisposableBean interface then destroy() method is called.

11) Call-custom destroy: if custom-destroy method is specified then it is called.

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.

```
<beans>

    <bean id="bar" class="com.act.Foo" init-method=”fooSetup” destroy=”fooTeardown”/>

</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.

25. What are the different types of bean injections?

There are two types of bean injections.

1. **By setter**
2. **By constructor**

Constructor Injection

Constructor-based DI is affected by invoking a constructor with a number of arguments, each representing a dependency.

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on a MovieFinder
    private MovieFinder movieFinder;

    // a constructor so that the Spring container can 'inject' a MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

```
package x.y;

public class Foo {

    public Foo(Bar bar, Baz baz) {
        // ...
    }
}
```

```
<beans>
    <bean name="foo" class="x.y.Foo">
        <constructor-arg>
            <bean class="x.y.Bar"/>
        </constructor-arg>
        <constructor-arg>
            <bean class="x.y.Baz"/>
        </constructor-arg>
    </bean>
</beans>
```

```
package examples;

public class ExampleBean {

    // No. of years to the calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

Constructor Argument Type Matching

The above scenario can *use type matching with simple types by explicitly specifying the type of the constructor argument using the 'type' attribute. For example:*

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

Constructor Argument Index

Constructor arguments can have their index specified explicitly by use of the index attribute. For example:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</bean>
```

Note that the *index is 0 based*.

Setter Injection

Setter-based Dependency Injection is realized by calling setter methods on your beans *after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.*

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can 'inject' a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

26. How is the Bean dependency resolved?

Bean dependency resolution generally happens as follows:

Step 1- The BeanFactory is created and initialized with a configuration which describes all the beans. (Most Spring users use a BeanFactory or ApplicationContext implementation that supports XML format configuration files.)

Step 2- Each bean has dependencies expressed in the form of properties, constructor arguments, or arguments to the static-factory method when that is used instead of a normal constructor. These dependencies will be provided to the bean, *when the bean is actually created*.

Step 3- Each property or constructor argument is either an actual definition of the value to set, or a reference to another bean in the container.

Step 4- Each property or constructor argument which is a value must be able to be converted from whatever format it was specified in, to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.

The Spring container validates the configuration of each bean as the container is created, including the validation that properties which are bean references are actually referring to valid beans. However, the bean properties themselves are not set until the bean *is actually created*.

For those beans that are singleton-scoped and set to be pre-instantiated (such as singleton beans in an ApplicationContext), creation happens at the time that the container is created, but otherwise this is only when the bean is requested.

27. What is Circular dependency?

Consider the scenario where you have class A, which requires an instance of class B to be provided via constructor injection, and class B, which requires an instance of class A to be provided via constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container will detect this circular reference at runtime, and throw a ***BeanCurrentlyInCreationException***

One possible solution to this issue is to edit the source code of some of your classes to be *configured via setters instead of via constructors*. Another solution is not to use constructor injection and stick to setter injection only.

In other words, while it should generally be avoided in all but the rarest of circumstances, it is possible to configure circular dependencies with setter injection.

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

28. Define <value> attribute?

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

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

```
<!-- typed as a java.util.Properties -->
<property name="properties">
    <value>
        jdbc.driver.className=com.mysql.jdbc.Driver
        jdbc.url=jdbc:mysql://localhost:3306/mydb
    </value>
</property>
```

The spring container is converting the text inside the <value/> element into a java.util.Properties instance using the JavaBeans PropertyEditor mechanism. This is a nice shortcut, and is one of a few places where the spring team do favour the use of the nested <value/> element over the 'value' attribute style.

29. Define idref element?

The idref element is simply an error-proof way to pass the *id* of another bean in the container (to a <constructor-arg/> or <property/> element).

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```

The above bean definition snippet is *exactly* equivalent (at runtime) to the following snippet:

```
<bean id="theTargetBean" class="..." />

<bean id="client" class="...">
    <property name="targetName" value="theTargetBean" />
</bean>
```

The main reason the first form is preferable to the second is that using the ***idref*** *tag allows the container to validate at deployment time that the referenced*, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the 'targetName' property of the 'client' bean. Any typo will only be discovered (with most likely fatal results) when the 'client' bean is actually instantiated. If the 'client' bean is a prototype bean, this typo (and the resulting exception) may only be discovered long after the container is actually deployed.

Additionally, if the bean being referred to is in the same XML unit, and the bean name is the bean *id*, the 'local' attribute may be used, which allows the XML parser itself to validate the bean id even earlier, at XML document parse time.

```
<property name="targetName">
   <!-- a bean with an id of 'theTargetBean' must exist; otherwise an XML exception will be thrown -->
   <idref local="theTargetBean"/>
</property>
```

30. Define ref attribute?

The ref element is the final element allowed inside a <constructor-arg/> or <property/> definition element. It is used to set the value of the specified property to be a reference to another bean managed by the container (a collaborator).

Specifying the target bean by using the bean attribute of the <ref/> tag is the most general form, and will allow creating a reference to any bean in the same container (whether or not in the same XML file), or parent container. The value of the 'bean' attribute may be the same as either the 'id' attribute of the target bean, or one of the values in the 'name' attribute of the target bean.

```
<ref bean="someBean"/>
```

Specifying the target bean by using the local attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the local attribute must be the same as the id attribute of the target bean. The XML parser will issue an error if no matching element is found in the same file.

```
<ref local="someBean"/>
```

Specifying the target bean by using the 'parent' attribute allows a reference to be created to a bean which is in a parent container of the current container. The value of the 'parent' attribute may be the same as either the 'id' attribute of the target bean, or one of the values in the 'name' attribute of the target bean, and the target bean must be in a parent container to the current one.

```
<!-- in the child (descendant) context -->
<bean id="accountService"  <-- notice that the name of this bean is the same as the name of the 'parent' bean
      class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
          <ref parent="accountService"/>  <-- notice how we refer to the parent bean
      </property>
    <!-- insert other configuration and dependencies as required as here -->
</bean>
```

31. What are inner Beans?

A <bean/> element inside the <property/> or <constructor-arg/> elements is used to define a so-called *inner bean*. An inner bean definition does not need to have any id or name defined, and it is best not to even specify any id or name value because the id or name value simply will be ignored by the container.

```
<bean id="outer" class="...">
  <!-- instead of using a reference to a target bean, simply define the target bean inline -->
  <property name="target">
    <bean class="com.example.Person"> <!-- this is the inner bean -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

Note - In the specific case of inner beans, the 'scope' flag and any 'id' or 'name' attribute are effectively ignored. Inner beans are *always* anonymous and they are *always* scoped as prototypes. Please also note that it is *not* possible to inject inner beans into collaborating beans other than the enclosing bean.

32. How Collection can be used with Beans?

The <list/>, <set/>, <map/>, and <props/> elements allow properties and arguments of the Java Collection type List, Set, Map, and Properties, respectively, to be defined and set.

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@example.org</prop>
        <prop key="support">support@example.org</prop>
        <prop key="development">development@example.org</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
        <value>a list element followed by a reference</value>
        <ref bean="myDataSource" />
    </list>
  </property>
  <!-- results in a setSomeMap(java.util.Map) call -->
  <property name="someMap">
    <map>
        <entry>
            <key>
                <value>an entry</value>
            </key>
            <value>just some string</value>
        </entry>
        <entry>
            <key>
                <value>a ref</value>
            </key>
            <ref bean="myDataSource" />
        </entry>
    </map>
  </property>
  <!-- results in a setSomeSet(java.util.Set) call -->
  <property name="someSet">
    <set>
        <value>just some string</value>
        <ref bean="myDataSource" />
    </set>
  </property>
</bean>
```

33. Define <null> element?

The <null/> element is used to handle null values. Spring treats empty arguments for properties and the like as empty Strings. The following XML-based configuration metadata snippet results in the email property being set to the empty String value ("")

```
<bean class="ExampleBean">
  <property name="email"><value/></property>
</bean>
```

This is equivalent to the following Java code: exampleBean.setEmail(""). The special <null> element may be used to indicate a null value. For example:

```
<bean class="ExampleBean">
  <property name="email"><null/></property>
</bean>
```

The above configuration is equivalent to the following Java code: exampleBean.setEmail(null).

34. What is p-namespace?

```
<bean name="john-modern"
    class="com.example.Person"
    p:name="John Doe"
    p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person">
    <property name="name" value="Jane Doe"/>
</bean>
```

this is telling Spring that it should include a property declaration

35. What are compound property names?

```
<bean id="foo" class="foo.Bar">
  <property name="fred.bob.sammy" value="123" />
</bean>
```

The foo bean has a fred property which has a bob property, which has a sammy property, and that final sammy property is being set to the value 123. In order for this to work, the fred property of foo, and the bob property of fred must not be null be non-null after the bean is constructed, or a NullPointerException will be thrown.

36. What is depends-on attribute?

The 'depends-on' attribute may be used to explicitly force one or more beans to be initialized before the bean using this element is initialized. Find below an example of using the 'depends-on' attribute to express a dependency on a single bean.

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />
```

If you need to express a dependency on multiple beans, you can supply a list of bean names as the value of the 'depends-on' attribute, with commas, whitespace and semicolons all valid delimiters, like so:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
  <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

Note - The 'depends-on' attribute at the bean definition level is used not only to specify an initialization time dependency, but also to specify the corresponding destroy time dependency (in the case of singleton beans only). Dependent beans that define a 'depends-on' relationship with a given bean will be destroyed first - prior to the given bean itself being destroyed. As a consequence, 'depends-on' may be used to control shutdown order too.

37. What is a Lazy-instantiated bean?

The default behavior for ApplicationContext implementations is to eagerly pre-instantiate all singleton beans at start-up. Pre-instantiation means that an ApplicationContext will eagerly create and configure all of its singleton beans as part of its initialization process.

However, there are times when this behavior is *not* what is wanted. If you do not want a singleton bean to be pre-instantiated when using an ApplicationContext, you can selectively control this by marking a bean definition as lazy-initialized. A lazily-initialized bean indicates to the IOC container whether or not a bean instance should be created at start-up or when it is first requested.

When configuring beans via XML, this lazy loading is controlled by the 'lazy-init' attribute on the <bean/> element; for example:

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

It is also possible to control lazy-initialization at the container level by using the 'default-lazy-init' attribute on the <beans/> element; for example:

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

38. What is Auto wiring?

The spring container is able to *autowire* relationships between collaborating beans. This means that it is possible to automatically let spring resolve collaborators (other beans) for your bean by inspecting the contents of the BeanFactory.

The autowiring functionality has five modes

- no
- byName
- byType
- constructor
- autodetect

When using XML-based configuration metadata, the autowire mode for a bean definition is specified by using the autowire attribute of the <bean/> element.

No - No autowiring at all. Bean references must be defined via a ref element. This is the default, and changing this is discouraged for larger deployments, since explicitly specifying collaborators gives greater control and clarity.

byname - Autowiring by property name. This option will inspect the container and look for a bean named exactly the same as the property which needs to be autowired. For example, if you have a bean definition which is set to autowire by name and it contains a *master* property (that is, it has a *setMaster (...)* method), spring will look for a bean definition named master, and use it to set the property.

byType - Allows a property to be autowired if there is exactly one bean of the property type in the container. If there is more than one, a fatal exception is thrown, and this indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set. If this is not desirable, setting the **dependency-check="objects"** attribute value specifies that an error should be thrown in this case.

Constructor - This is analogous to *byType*, but applies to constructor arguments. If there isn't exactly one bean of the constructor argument type in the container, a fatal error is raised.

Autodetect - Chooses *constructor* or *byType* through introspection of the bean class. If a default constructor is found, the *byType* mode will be applied.

<bean id="bar" class="com.act.Foo" **autowire="autowire type"/>**

39. How can be a Bean excluded from being available for autowiring

We can totally exclude a bean from being an autowire candidate using 'autowire-candidate' attribute of the <bean/> element can be set to 'false'. This has the effect of making the container totally exclude that specific bean definition from being available to the autowiring infrastructure.

<bean id="bar" class="com.act.Foo" **autowire-candidate="false"/>**

Another option is to limit autowire candidates based on pattern-matching against bean names. The top-level <beans/> element accepts one or more patterns within its 'default-autowire-candidates' attribute. For example,

to limit autowire candidate status to any bean whose name ends with *'Repository'*, provide a value of '*Repository'. To provide multiple patterns, define them in a comma-separated list.

<beans **default-autowire-candidate ="”***Repository,*Stub**”>**

…..

</beans>

Note that an explicit value of 'true' or 'false' for a bean definition's 'autowire-candidate' attribute always takes precedence, and for such beans, the pattern matching rules will not apply.

40. How the Bean dependencies can be checked?

The Spring IoC container also has the ability to check for the existence of unresolved dependencies of a bean deployed into the container.

This feature is sometimes useful when you want to ensure that all properties (or all properties of a certain type) are set on a bean. Of course, in many cases a bean class will have default values for many properties, or some properties do not apply to all usage scenarios, so this feature is of limited use. Dependency checking can also be enabled and disabled per bean, just as with the autowiring functionality. The default is to *not* check dependencies.

This can be done by using 'dependency-check' attribute in a bean definition.

Mode	Description
none	No dependency checking. Properties of the bean which have no value specified for them are simply not set.
simple	Dependency checking is performed for primitive types and collections (everything except collaborators).
object	Dependency checking is performed for collaborators only.
all	Dependency checking is done for collaborators, primitive types and collections.

41. What are the different types of Bean scopes available in spring?

The scope of bean represent the life cycle of Bean.

Scope	Description
singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
global session	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

The global session scope is similar to the standard HTTP Session scope and really only makes sense in the context of portlet-based web applications. The portlet specification defines the notion of a global Session that is shared amongst all of the various portlets that make up a single portlet web application. Beans defined at the global session scope are scoped (or bound) to the lifetime of the global portlet Session.

42. What is <aop: scoped-proxy/>?

If you want to inject a (for example) HTTP request scoped bean into another bean, you will need to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object, but that is smart enough to be able to retrieve the real, target object from the relevant scope (for example a HTTP request) and delegate method calls onto the real object.

You need the following, correct and complete, configuration when injecting request-, session-, and globalSession-scoped beans into collaborating objects:

```
<!-- a HTTP Session-scoped bean exposed as a proxy -->
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

      <!-- this next element effects the proxying of the surrounding bean -->
      <aop:scoped-proxy/>
</bean>
```

Note - You *do not* need to use the <aop:scoped-proxy/> in conjunction with beans that are scoped as singletons or prototypes. It is an error to try to create a scoped proxy for a singleton bean (and the resulting BeanCreationException will certainly set you straight in this regard).

To create such a proxy, you need only to insert a child <aop:scoped-proxy/> element into a scoped bean definition (you may also need the CGLIB library on your classpath so that the container can effect class-based proxying

Note: CGLIB proxies will only intercept public method calls! Do not call non-public methods on such a proxy; they will not be delegated to the scoped target object.

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

43. How you will create your own custom scope?

Scopes are defined by the org.springframework.beans.factory.config.Scope interface. This is the interface that you will need to implement in order to integrate your own custom scope(s) into the Spring container

The Scope interface has four methods dealing with getting objects from the scope, removing them from the scope and allowing them to be 'destroyed' if needed.

First Method

The first method should return the object from the underlying scope. The session scope implementation for example will return the session-scoped bean (and if it does not exist, return a new instance of the bean, after having bound it to the session for future reference).

Object get(String name, ObjectFactory objectFactory)

Second Method

The second method should remove the object from the underlying scope.

Object remove (String name)

The object should be returned (you are allowed to return null if the object with the specified name wasn't found)

Third Method

The third method is used to register callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed.

void registerDestructionCallback(String name, Runnable destructionCallback)

Fourth Method

The last method deals with obtaining the conversation identifier for the underlying scope. This identifier is different for each scope. For a session for example, this can be the session identifier.

String getConversationId()

After you have written and tested one or more custom Scope implementations, you then need to make the Spring container aware of your new scope(s). The central method to register a new Scope with the Spring container is declared on the ConfigurableBeanFactory interface (implemented by most of the concrete BeanFactory implementations that ship with Spring); this central method is displayed below:

void registerScope(String scopeName, Scope scope);

The first argument to the registerScope(..) method is the unique name associated with a scope; examples of such names in the Spring container itself are 'singleton' and 'prototype'. The second argument to the registerScope(..) method is an actual instance of the custom Scope implementation that you wish to register and use.

```
// note: the ThreadScope class does not ship with the Spring Framework
Scope customScope = new ThreadScope();
beanFactory.registerScope("thread", customScope);
```

<bean id="..." class="..." **scope="thread"**/>

44. What is ApplicationContext?

The ApplicationContext builds on top of the BeanFactory and inherits all the basic features of the Spring Framework. Apart from the basic features, ApplicationContext provides additional features i.e.

- MessageSource, providing access to messages in i18n-style.
- *Access to resources*, such as URLs and files.
- *Event propagation* to beans implementing the ApplicationListener interface.
- *Loading of multiple (hierarchical) contexts*, allowing each to be focused on one particular layer, for example the web layer of an application.

The BeanFactory is particularly useful in low memory situations like in an Applet where having the whole API would be memory consuming might be critical and a few extra kilobytes might make a difference. It provides the basic spring framework features and does not bring all the excess baggage that ApplicationContext has. ApplicationContext helps the user to use spring in a framework oriented way while the BeanFactory offers a programmatic approach.

45. How you will use Internationalization using MessageSources?

The ApplicationContext interface extends an interface called MessageSource, and therefore provides messaging (i18n or internationalization) functionality.

String getMessage(String code, Object[] args, String default, Locale loc)

String getMessage(String code, Object[] args, Locale loc)

String getMessage(MessageSourceResolvable resolvable, Locale locale)

When an ApplicationContext gets loaded, it automatically searches for a MessageSource bean defined in the context. The bean has to have the name 'messageSource'. If such a bean is found, all calls to the methods described above will be delegated to the message source that was found. If no message source was found, the ApplicationContext attempts to see if it has a parent containing a bean with the same name. If so, it uses that bean as the MessageSource. If it can't find any source for messages, an empty DelegatingMessageSource will be instantiated in order to be able to accept calls to the methods defined above.

```
<beans>
  <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>format</value>
        <value>exceptions</value>
        <value>windows</value>
      </list>
    </property>
  </bean>
</beans>
```

This assumes you have three resource bundles defined on your classpath called format, exceptions and windows.

```
# in 'format.properties'
message=Alligators rock!
```

```
# in 'exceptions.properties'
argument.required=The '{0}' argument is required.
```

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

   MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");

   String message = resources.getMessage("message", null, "Default", null);

   System.out.println(message);

}
```

46. What are the different types of events related to Listeners?

Event handling / Listener in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. If a bean which implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean will be notified. Essentially, this is the standard *Observer* design pattern.

Event	Explaination
ContextRefreshedEvent	Published when the ApplicationContext is initialized or refreshed e.g. using the refresh() method on the ConfigurableApplicationContext interface.
ContextStartedEvent	Published when the ApplicationContext is started, using the start() method on the ConfigurableApplicationContext interface
ContextStoppedEvent	Published when the ApplicationContext is stopped, using the stop() method on the ConfigurableApplicationContext interface
ContextClosedEvent	Published when the ApplicationContext is closed, using the close() method on the ConfigurableApplicationContext interface. "Closed" here means that all singleton beans are destroyed. A closed context has reached its end of life; it cannot be refreshed or restarted.
RequestHandledEvent	A web-specific event telling all beans that an HTTP request has been serviced (this will be published *after* the request has been finished). Note that this event is only applicable for web applications using Spring's DispatcherServlet.

Implementing custom events can be done as well. Simply call the publishEvent() method on the ApplicationContext, specifying a parameter which is an instance of your custom event class implementing ApplicationEvent. Event listeners receive events synchronously. This means the publishEvent() method blocks until all listeners have finished processing the event (it is possible to supply an alternate event publishing strategy via a ApplicationEventMulticaster implementation).

```
<bean id="emailer" class="example.EmailBean">
  <property name="blackList">
    <list>
      <value>black@list.org</value>
      <value>white@list.org</value>
      <value>john@doe.org</value>
    </list>
  </property>
</bean>

<bean id="blackListListener" class="example.BlackListNotifier">
  <property name="notificationAddress" value="spam@list.org"/>
</bean>
```

Now, let's look at the actual classes:

```
public class EmailBean implements ApplicationContextAware {

    private List blackList;
    private ApplicationContext ctx;

    public void setBlackList(List blackList) {
        this.blackList = blackList;
    }

    public void setApplicationContext(ApplicationContext ctx) {
        this.ctx = ctx;
    }

    public void sendEmail(String address, String text) {
        if (blackList.contains(address)) {
            BlackListEvent event = new BlackListEvent(address, text);
            ctx.publishEvent(event);
            return;
        }
        // send email...
    }
}
```

```
public class BlackListNotifier implements ApplicationListener {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof BlackListEvent) {
            // notify appropriate person...
        }
    }
}
```

47. What is IOC?

The basic concept of the Inversion of Control pattern (also known as dependency injection) is that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (in the case of the spring framework, the IOC container) is then responsible for hooking it all up.

48. What are the benefits of IOC (Dependency Injection)?

Benefits of IOC (Dependency Injection) are as follows:

- Minimizes the amount of code in your application. With IOC containers you do not care about how services are created and how you get references to the ones you need. You can also easily add additional services by adding a new constructor or a setter method with little or no extra configuration.

- Make your application more testable by not requiring any singletons or JNDI lookup mechanisms in your unit test cases. IOC containers make unit testing and switching implementations very easy by manually allowing you to inject your own objects into the object under test.

- Loose coupling is promoted with minimal effort and least intrusive mechanism. The factory design pattern is more intrusive because components or services need to be requested explicitly whereas in IOC the dependency is injected into requesting piece of code. Also some containers promote the design to interfaces not to implementations design concept by encouraging managed objects to implement a well-defined service interface of your own.

- IOC containers support eager instantiation and lazy loading of services. Containers also provide support for instantiation of managed objects, cyclical dependencies, life cycles management, and dependency resolution between managed objects etc.

49. What is the difference between Bean Factory and Application Context?

On the surface, an application context is same as a bean factory. But application context offers much more..

- Application contexts provide a means for resolving text messages, including support for i18n of those messages.
- Application contexts provide a generic way to load file resources, such as images.
- Application contexts can publish events to beans that are registered as listeners.
- Certain operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context.
- ResourceLoader support: Spring's Resource interface us a flexible generic abstraction for handling low-level resources. An application context itself is a ResourceLoader, Hence provides an application with access to deployment-specific Resource instances.
- MessageSource support: The application context implements MessageSource, an interface used to obtain localized messages, with the actual implementation being pluggable

50. What are ORM's Spring supports ?

Spring supports the following ORM's :

- Hibernate
- iBatis

- JPA (Java Persistence API)
- TopLink
- JDO (Java Data Objects)
- OJB

51. How to integrate Spring and Hibernate?

Spring and Hibernate can integrate using Spring's SessionFactory called LocalSessionFactory. The integration process is of 3 steps.

- Configure Hibernate mappings.
- Configure Hibernate properties.
- Wire dependant object to SessionFactory.

52. **What are the ways to access Hibernate using Spring?**

There are two ways to access Hibernate from Spring:

- Through Hibernate Template.
- Subclassing HibernateDaoSupport
- Extending HibernateDaoSupport and Applying an AOP Interceptor

Spring JDBC Questions

Version 1.0

Sanjeev Kumar Singh

53. How can you configure a bean to get DataSource from JNDI?

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"> <value>java:comp/env/jdbc/myDatasource</value>
 </property>
</bean>
```

54. How can you create a DataSource connection pool?

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driver">
            <value>${db.driver}</value>
      </property>
      <property name="url">
           <value>${db.url}</value>
      </property>
      <property name="username">
          <value>${db.username}</value>
      </property>
      <property name="password">
         <value>${db.password}</value>
      </property>
</bean>
```

55. 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.

56. What is JdbcTemplate class used for in Spring's JDBC API?

The JdbcTemplate class is the central class in the JDBC core package. It simplifies the use of JDBC since it handles the creation and release of resources. This class executes SQL queries, update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions defined in the hierarchy of org.springframework.dao package. Code using the JdbcTemplate only need to implement callback interfaces, giving them a clearly defined contract. The PreparedStatementCreator callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The JdbcTemplate can be used within a DAO implementation via direct instantiation with a DataSource reference, or be configured in a Spring IOC container and given to DAOs as a bean reference.

57. 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.

JdbcTemplate template = new **JdbcTemplate**(myDataSource);

```
public class StudentDaoJdbc implements StudentDao {
        private JdbcTemplate jdbcTemplate;

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

XML Configuration will be as follows

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

58. How do you write data to database in spring using JdbcTemplate?

The JdbcTemplate uses several of these callbacks when writing data to the database. There are two simple interfaces.

1. **PreparedStatementCreator -** This interface creates a PreparedStatement given a connection, provided by the JdbcTemplate class. Implementations are responsible for providing SQL and any necessary parameters.
2. The ResultSetExtractor interface extracts values from a ResultSet
3. BatchPreparedStatementSetter interface execute the SQL batch

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

final int count = 2000;
final List<String> firstNames = new ArrayList<String>(count);
final List<String> lastNames = new ArrayList<String>(count);
for (int i = 0; i < count; i++) {
  firstNames.add("First Name " + i);
  lastNames.add("Last Name " + i);
}
jdbcTemplate
    .batchUpdate(
        "insert into customer (id, first_name, last_name, last_login, comments) values (?, ?, ?, ?, ?)",
        new BatchPreparedStatementSetter() {
          public void setValues(PreparedStatement ps, int i) throws SQLException {
            ps.setLong(1, i + 10);
            ps.setString(2, firstNames.get(i));
            ps.setString(3, lastNames.get(i));
            ps.setNull(4, Types.TIMESTAMP);
            ps.setNull(5, Types.CLOB);
          }
          public int getBatchSize() {
            return count;
          }
        });
}
```

59. Explain about PreparedStatementCreator?

PreparedStatementCreator is one of the most common used interfaces for writing data to database. The interface has one method createPreparedStatement().

PreparedStatement **createPreparedStatement**(Connection conn) throws SQLException;

When this interface is implemented, another interface **SqlProvider** is also implemented which has a method called **getSql**() which is used to provide sql strings to JdbcTemplate.

60. Explain about BatchPreparedStatementSetter?

If the user what to update more than one row at a shot then he can go for **BatchPreparedStatementSetter**. This interface provides two methods

setValues(PreparedStatement ps, int i) throws SQLException;
int getBatchSize();

The getBatchSize() tells the JdbcTemplate class how many statements to create. And this also determines how many times setValues() will be called.

61. Explain about RowCallbackHandler and why it is used?

In order to navigate through the records we generally go for ResultSet. But spring provides an interface that handles this entire burden and leaves the user to decide what to do with each row. The interface provided by spring is RowCallbackHandler. There is a method processRow() which needs to be implemented so that it is applicable for each and every row.

void **processRow**(java.sql.ResultSet rs);

62. What is NamedParameterJdbcTemplate class used for in Spring's JDBC API?

The NamedParameterJdbcTemplate class adds support for programming JDBC statements using named parameters (as opposed to programming JDBC statements using only classic placeholder ('?') arguments. The NamedParameterJdbcTemplate class wraps a JdbcTemplate, and delegates to the wrapped JdbcTemplate to do much of its work.

```
public void insertForum(Forum forum) {
    String query = "INSERT INTO FORUMS (FORUM_ID, FORUM_NAME,
      FORUM_DESC) VALUES (:forumId,:forumName,:forumDesc)";
    Map namedParameters = new HashMap();
    namedParameters.put("forumId", Integer.valueOf(forum.getForumId()));
    namedParameters.put("forumName", forum.getForumName());
    namedParameters.put("forumDesc", forum.getForumDesc());
    namedParameterJdbcTemplate.update(query, namedParameters);
}
```

63. Name the JDBC Core classes to control basic JDBC processing and error handling in Spring's JDBC API?

The JDBC Core classes to control basic JDBC processing and error handling in Spring's JDBC API are : JdbcTemplate

- NamedParameterJdbcTemplate
- SimpleJdbcTemplate (Java5)
- DataSource
- SQLExceptionTranslator

Spring –Transaction Management Questions

Version 1.0

Sanjeev Kumar Singh

64. What are the benefits of Spring Transaction Management?

There are following benefits using spring transaction management.

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
- Supports declarative transaction management.
- Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- Integrates very well with spring's various data access abstractions.

As well as the J2EE developers have had two choices for transaction management: GLOBAL or LOCAL transactions.

Global transactions are managed by the application server, using the Java Transaction API (JTA). Global transaction uses JTA that is very cumbersome API and its UserTransactions is sourced from JNDI it means we need to use both JNDI and JTA. Now the JTA is available in only Application Server envionement . Previously, the preferred way to use global transactions was via EJB CMT (Container Managed Transaction): CMT is a form of declarative transaction management (as distinguished from programmatic transaction management). EJB CMT removes the need for transaction-related JNDI lookups - The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in EJBs.

Local transactions are resource-specific: the most common example would be a transaction associated with a JDBC connection. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Another downside is that local transactions tend to be invasive to the programming model.

Spring resolves these problems. It enables application developers to use a CONSISTENT programming model IN ANY ENVIRONMENT. You write your code once, and it can benefit from different transaction management strategies in different environments.

65. How do you define the transaction strategy?

Spring Transaction Management API contains a transaction strategy defined by org.springframework.transaction.PlatformTransactionManager interface

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

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

- PlatformTransactionManager is an INTERFACE, and can thus be easily mocked or stubbed as necessary.
- Nor is it tied to a lookup strategy such as JNDI
- PlatformTransactionManager implementations are defined like any other object (or bean) in the Spring Framework's IOC container.
- The TransactionException that can be thrown by any of the PlatformTransactionManager interface's methods is UNCHECKED (i.e. it extends the java.lang.RuntimeException class).

The getTransaction (...) method returns a TransactionStatus object, depending on a TransactionDefinition parameter. The returned TransactionStatus might represent a new or existing transaction. Regardless of whether you opt for declarative or programmatic transaction management in spring, defining the correct PlatformTransactionManager implementation is absolutely essential.

66. What is the role of TransactionDefinition?

Basically TransactionDefinition is an interface, which specifies.

- Isolation: the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?

- Propagation: normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case); or suspending the existing transaction and creating a new transaction. Spring offers all of the transaction propagation options familiar from EJB CMT.

- Timeout: how long this transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).

- Read-only status: a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

67. What is the role of TransactionStatus?

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

```
public interface TransactionStatus extends SavepointManager {

    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();

}
```

68. How do you Configure JTA with spring transaction API?

Step 1 – Need to define a Datasource

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

Step 2 – Create the Transaction Manager which will be responsible for executing the transaction

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

Step 3 – Declare the bean details in 'dataAccessContext-jta.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:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

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

  <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />

  <!-- other <bean/> definitions here -->

</beans>
```

69. How do you Configure Hibernate with spring transaction API?

Step 1 – Need to define a Datasource

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

Step 2 – Create the Transaction Manager which will be responsible for executing the transaction

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

Step 3 - Declare the bean details in ApplicationContext-hibernate.xml

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
          hibernate.dialect=${hibernate.dialect}
        </value>
  </property>
</bean>

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

70. What is Programmatic Transaction Model?

From the Programmatic Transaction model the developer is responsible for obtaining a transaction from the transaction manager, starting the transaction, committing the transaction, and — if an exception occurs — rolling back the transaction.

The Spring Framework has two ways of implementing the Programmatic Transaction model. One way is through the Spring **TransactionTemplate**, and the other is by using a spring **platform transaction manager** directly.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

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

First Approach - Using Transaction Manager

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

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

The anonymous DefaultTransactionDefinition class contains details about the transaction and its behavior, including the transaction name, isolation level, propagation mode (transaction attribute), and transaction timeout (if any). In this case, simply using the default values, which are an empty string for the name, the default isolation level for the underlying DBMS (usually READ_COMMITTED), PROPAGATION_REQUIRED for the transaction attribute, and the default timeout of the DBMS

Second Appraoch - Using Transaction Template

The TransactionTemplate adopts the same approach as other Spring templates such as the JdbcTemplate.

If the transaction returns some object

```
public class SimpleService implements Service {

    // single TransactionTemplate shared amongst all methods in this instance
    private final TransactionTemplate transactionTemplate;

    // use constructor-injection to supply the PlatformTransactionManager
    public SimpleService(PlatformTransactionManager transactionManager) {
        Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object someServiceMethod() {
        return transactionTemplate.execute(new TransactionCallback() {

            // the code in this method executes in a transactional context
            public Object doInTransaction(TransactionStatus status) {
                updateOperation1();
                return resultOfUpdateOperation2();
            }
        });
    }
}
```

If there is no return value, use the convenient TransactionCallbackWithoutResult class via an anonymous class like so

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

Code within the callback can roll the transaction back by calling the setRollbackOnly() method on the supplied TransactionStatus object.

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        try {
            updateOperation1();
            updateOperation2();
        } catch (SomeBusinessExeption ex) {
            status.setRollbackOnly();
        }
    }
});
```

71. What is Declarative Transaction Model

The Declarative Transaction model, otherwise known as *Container Managed Transactions* (CMT), is the most common transaction model in the Java platform. In this model, the container environment takes care of starting, committing, and rolling back the transaction. The developer is responsible only for specifying the transactions' behavior.

Spring used @Transactional annotation to specify the transaction. The container will not automatically roll back a transaction on a checked exception when you use the Declarative Transaction model

```
// the service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

  Foo getFoo(String fooName);

  Foo getFoo(String fooName, String barName);

  void insertFoo(Foo foo);

  void updateFoo(Foo foo);
}
```

The @Transactional annotation may be placed before an interface definition, a method on an interface, a class definition, or a PUBLIC method on a class.

Presence of the `@Transactional` annotation is not enough to actually turn on the transactional behaviour , need to add `<tx:annotation-driven/>` element

Now need to add transaction behavior in XML file

```
<!-- this is the service object that we want to make transactional -->
<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- enable the configuration of transactional behavior based on annotations -->
<tx:annotation-driven transaction-manager="txManager"/>

<!-- a PlatformTransactionManager is still required -->
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <!-- (this dependency is defined somewhere else) -->
  <property name="dataSource" ref="dataSource"/>
</bean>
```

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the DefaultFooService class is annotated at the class level with the settings for a read-only transaction, but the @Transactional annotation on the updateFoo(Foo) method in the same class takes precedence over the transactional settings defined at the class level.

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {
    // do something
  }

  // these settings have precedence for this method
  @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
  public void updateFoo(Foo foo) {
    // do something
  }
}
```

72. What are the attributes available for <tx:annotation-driven/> ?

Attribute	Required?	Default	Description
transaction-manager	No	transactionManager	The name of transaction manager to use. Only required if the name of the transaction manager is not transactionManager, as in the example above.
proxy-target-class	No		Controls what type of transactional proxies are created for classes annotated with the @Transactional annotation. If "proxy-target-class" attribute is set to "true", then class-based proxies will be created. If "proxy-target-class" is "false" or if the attribute is omitted, then standard JDK interface-based proxies will be created
order	No		Defines the order of the transaction advice that will be applied to beans annotated with <code>@Transactional</code>.

73. What are the @transactional annotation properties?

Property	Type	Description
propagation	enum: Propagation	optional propagation setting
isolation	enum: Isolation	optional isolation level
readOnly	Boolean	read/write vs. read-only transaction
timeout	int (in seconds granularity)	the transaction timeout
rollbackFor	an array of Class objects, which must be derived from Throwable	an optional array of exception classes which **must** cause rollback
rollbackForClassname	an array of class names. Classes must be derived from Throwable	an optional array of names of exception classes that **must** cause rollback
noRollbackFor	an array of Class objects, which must be derived from Throwable	an optional array of exception classes that **must not** cause rollback.
noRollbackForClassname	an array of String class names, which must be derived from Throwable	an optional array of names of exception classes that **must not** cause rollback

74. What different type of transaction attributes or propagation properties supported by Spring?

There are 6 types of propagation supported by spring:

- **Required** – *Support a current transaction. If not exist then it will create a new transaction* (If the Required transaction attribute is specified for methodA() and methodA() is invoked under the scope of an existing transaction, the existing transaction scope will be used. Otherwise, methodA() will start a new transaction. If the transaction is started by methodA(), then it must also be terminated (committed or rolled back) by methodA())
- **Mandatory** - *Support a current transaction. If there is no current transaction exist then it will throw an exception* (If the Mandatory transaction attribute is specified for methodA () and methodA() is invoked under an existing transaction's scope, the existing transaction scope will be used. However, if methodA() is invoked without a transaction context, then a TransactionRequiredException will be thrown, indicating that a transaction must be present before methodA() is invoked.).
- **RequiresNew** – *Create a new transaction and suspend the current transaction if exist* (If the RequiresNew transaction attribute is specified for methodA() and methodA() is invoked with or without a transaction context, a new transaction will always be started (and terminated) by methodA(). This means that if methodA() is invoked within the context of another transaction (called Transaction1 for example), Transaction1 will be suspended and a new transaction (called Transaction2) will be started. Once methodA() ends, Transaction2 is then either committed or rolled back, and Transaction1 resumes)
- **Supports** – Support the current transaction and if no transaction exists then it will execute without creating the transaction (Supports transaction attribute is specified for methodA () and methodA() is invoked within the scope of an existing transaction, methodA() will execute under the scope of that transaction. However, if methodA() is invoked without a transaction context, then no

transaction will be started. This attribute is primarily used for read-only operations to the database.)

- **NotSupported** – Do not support the current transaction, always execute non transactionally (The NotSupported transaction attribute specifies that the method being called will not use or start a transaction, regardless if one is present. If the NotSupported transaction attribute is specified for methodA() and methodA() is invoked in context of a transaction, that transaction is suspended until methodA() ends. When methodA() ends, the original transaction is then resumed.).
- **Never** – Do not support the current transaction and if current transaction exists it will throw an exception (if a transaction context exists when a method is called using the Never transaction attribute, an exception is thrown indicating that a transaction is not allowed when you invoke that method.).

75. What are different Isolation levels provided by spring?

There are following five Isolation level provided by spring.

- ISOLATION_DEFAULT
- ISOLATION_READ_COMMITTED
- ISOLATION_READ_UNCOMMITTED
- ISOLATION_REPEATABLE_READ
- ISOLATION_SERIALIZABLE

Spring DAO Questions

Version 1.0

Sanjeev Kumar Singh

76. How does spring support DAO in hibernate?

    The Data Access Object (DAO) support in spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a consistent way.

    Spring's HibernateDaoSupport class is a convenient super class for Hibernate DAOs. It has handy methods you can call to get a Hibernate Session, or a SessionFactory. The most convenient method is getHibernateTemplate(), which returns a HibernateTemplate. This template wraps Hibernate checked exceptions with runtime exceptions, allowing your DAO interfaces to be Hibernate exception-free. Example:

```
public class UserDAOHibernate extends HibernateDaoSupport {
public User getUser(Long id) {
return (User) getHibernateTemplate().get(User.class, id);
}
public void saveUser(User user) {
getHibernateTemplate().saveOrUpdate(user);
if (log.isDebugEnabled()) {
log.debug("userId set to: " + user.getID());
}
}
public void removeUser(Long id) {
Object user = getHibernateTemplate().load(User.class, id);
getHibernateTemplate().delete(user);
}
```

77. Explain the Spring's DataAccessException?

    Spring's org.springframework.dao.DataAccessException extends the NestedRuntimeException which in turn extends and RuntimeException. Hence DataAcccessException is a RuntimeException so there is no need to declare it in the method signature.

78. What is the exception class related to all exceptions that are thrown in Spring Application?

    DataAccessException - org.springframework.dao.DataAccessException

79. What is DataAccessException?

    DataAccessException is Unchecked Runtime Exception

Spring Configuration Questions

Version 1.0

Sanjeev Kumar Singh

81. How do you configure Hibernate with spring?

There are following steps need to perform to integration of hibernate with spring.

1. Create the Datasource
2. Create the required hbm files

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"><value>org.hsqldb.jdbcDriver</value></property>
        <property name="url"><value>jdbc:hsqldb:mem:test</value></property>
        <property name="username"><value>sa</value></property>
        <property name="password"><value></value></property>
    </bean>

    <!-- Hibernate SessionFactory -->
    <bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource"><ref local="dataSource"/></property>
        <property name="mappingResources">
            <list>
                <value>MyRecord.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">net.sf.hibernate.dialect.HSQLDialect</prop>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
        </props>
        </property>
    </bean>

    <!-- Transaction manager for a single Hibernate SessionFactory (alternative to JTA) -->
    <bean id="transactionManager" class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
    </bean>

    <bean id="myRecordDAO" class="com.shoesobjects.dao.MyRecordDAOHibernateWithSpring">
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
    </bean>
</beans>
```

3. Create the sessionFactory Bean
4. Create the transactionManager
5. Create your respective DAO and extends with HibernateDaoSupport

```
package com.shoesobjects.dao;

import com.shoesobjects.MyRecord;
import org.springframework.orm.hibernate.support.HibernateDaoSupport;

import java.util.List;

public class MyRecordDAOHibernateWithSpring extends HibernateDaoSupport implements MyRecordDAO {

    public MyRecord getRecord(Long id) {
        return (MyRecord) getHibernateTemplate().get(MyRecord.class, id);
    }

    public List getRecords() {
        return getHibernateTemplate().find("from MyRecord");
    }

    public void saveRecord(MyRecord record) {
        getHibernateTemplate().saveOrUpdate(record);
    }

    public void removeRecord(Long id) {
        Object record = getHibernateTemplate().load(MyRecord.class, id);
        getHibernateTemplate().delete(record);
    }
}
```

82. How you will configute HibernateTemplate?

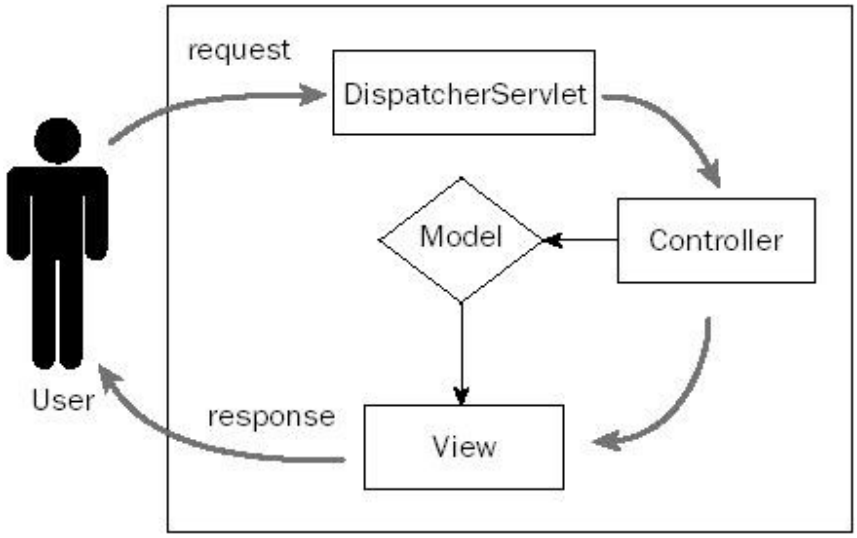
```
<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
<property name="sessionFactory">
<ref bean="mySessionFactory"/>
</property>
</bean>

<bean id="hibernateDao" class="HibernateAccessDao">
<property name="hibernateTemplate">
<ref bean="hibernateTemplate"/>
</property>
</bean>
```

Spring MVC Questions

Version 1.0

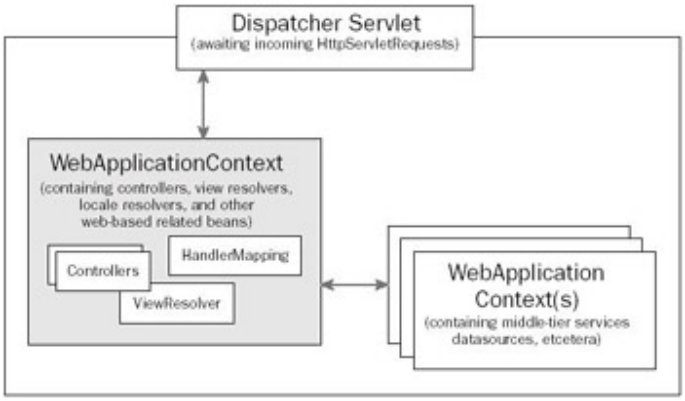
Sanjeev Kumar Singh

83. How Spring MVC process the client request?
    1. Client request to access a resource in a web application
    2. This request is handled by front controller known as dispatcher Servlet
    3. The Dispatcher Servlet intercepts the request and identify the appropriate handler to serve this request in <Servlet-name>-servlet.xml
    4. With the help of handler adapter the dispatcher Servlet dispatch the request to appropriate controller.
    5. The controller process the client request and return the Model and the view in format of ModelAndView object to the dispatcher Servlet.
    6. Now the dispatcher Servlet resolves the corresponding view with the help of View Resolver.
    7. After that the view is returned to the client.



84. What is Dispatcher Servlet in spring MVC?

    This is the entry point of for any request. This will inspect it and determine what controller it needs to execute the given request. It dispatches requests to controllers and takes care of rendering the response, based on the model and view information the controller returns.



When the configuration is setup the dispatcher servlet work as the central servlet to handle the request and response. Spring uses a special type of context WebApplicationContext (similar to application context) which is connected with the Dispatcher servlet to provide the required functionality, as well as it manages all web related components such as controllers, views , mapping between url and interceptors.

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
       org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.*</url-pattern>
</servlet-mapping>
```

85. What is WebApplicationContext?

    The WebApplicationContext contains all web-related beans responsible for making your logic accessible to users. These may include beans to handle multilingual websites and beans defining mappings from incoming requests to controllers. The WebApplicationContext will be initialized by the DispatcherServlet. When using default values, it will be of class org.springframework.web.context.support. XmlWebApplicationContext.

    The WebApplicationContext is a child context of the ApplicationContext, which will contain middle-tier services such as transactional service layer objects, data sources, and data access objects.XmlWebApplicationContext is used by the dispatcher servlet to handle the flow of the application. When a request arrived at the dispatcher servlet, it identifies the request with the help of webapplicationcontext to which controller process this request. Basically the webapplicationcontext contains lots of the bean to process the request which is dispatched by the dispatcher servlet.

86. What are handlers mapping?

When the Client send request, this request reaches to the ***Dispatcher Servlet***, the ***Dispatcher Servlet*** tries to find the appropriate ***Handler Mapping*** Object to map between the Request and the Handling Object. Basically, Handler mappings determine the path of execution for a request a user has issued.

HandlerMapping implements the ordered interface so if more than one mapping is provided for one URL than it will take that handler which will be the first position after ordered sorting.

If any handler is not defined in the context then spring uses the BeanNameUrlHandlerMapping.

Spring provides the following handler mapping techniques:

```
<beans>
  <bean name="/showAllMails.jsp"
class="com.javabeat.net.ShowAllMailsController">
  </bean>

  <bean name="/composeMail.jsp"
  class="com.javabeat.net.ComposeMailController">
  </bean>

  <bean name="/ deleteMail.jsp"
  class="com.javabeat.net.DeleteMailController">
  </bean>

</beans>
```

- BeanNameUrlHandlerMapping - it is used to map the URL that comes from the Clients directly to the Bean Object. Actually the Bean is nothing but a Controller object. For Example the Requested URL is http://myserver.com/eMail/showAllMails. Now BeanNameUrlHandlerMapping find the appropriate bean defined above and the request will be dispatched to respective controller.

  To enable this mapping need to define a bean a mapping in configuration file.

```
<beans> <bean id="beanNameUrl"
          class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
</beans>
```

- SimpleUrlHandlerMapping - This is the Simplest of all the Handler Mappings as it directly maps the Client Reqest to some Controller object. Consider the following Configuration File,

```
<bean id="simpleUrlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
                <props>
                        <prop key="/showAllMails.jsp">showController</prop>
                        <prop key="/composeMail.jsp">composeController</prop>
                        <prop key="/deleteMail.jsp">deleteController</prop>
                </props>
        </property>

</bean>
```

- CommonsPathMapUrlHandlerMapping - Using the CommonsPathMapUrlHandlerMapping we don't need to create the mapping between controller and URL in the ApplicationContext. Here we set the mapping attribute in the controller itself. In this case we put a annotation in controller class.

```
*
 * @@org.springframework.web.servlet.handler.metadata.PathMap("/showGenres.html")
 */
```

public GenreController extends AbstractController {… ……}

It is useful when we want to use the controller in multiple web application and we don't want to create different mappings in the different ApplicationContext files.

- ControllerClassNameHandlerMapping – It is a special type of handler which works on different way. To use this controller the name of our controller class should be suffixed by Controller. When the request is generated and to get the appropriate handler the convention is to take the short name of the Class, remove the 'Controller' suffix if it exists and return the remaining text, lowercased, as the mapping, with a leading /. For example:

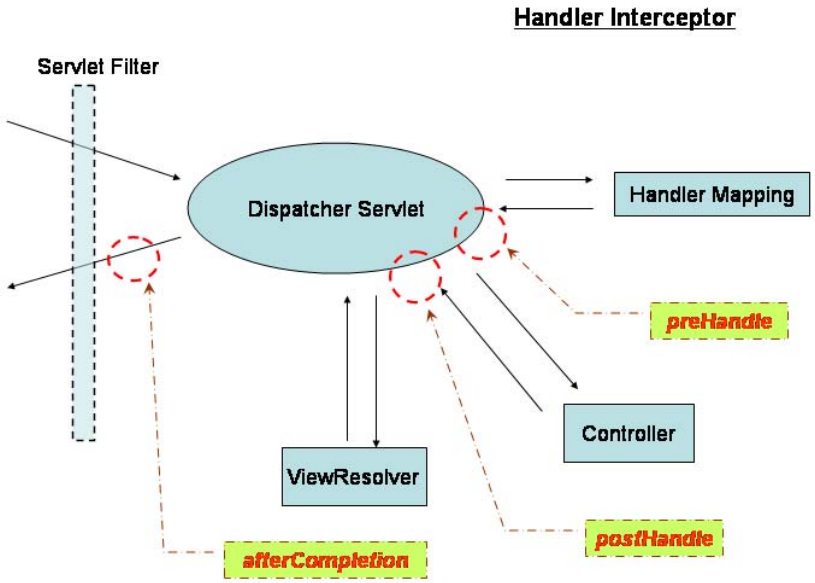
  WelcomeController -> /welcome*
  HomeController -> /home*

87. What are handlers mapping?

Once the DispatcherServlet finds the actual handler mapping to handle the request, that request is dispatched to that handler but this is achieved in the form of Handler Adapters. Servlet chooses the appropriate Handler Mapping, the Request is then forwarded to the Controller object that is defined in the Configuration File. This is the default case. And this so happens because the Default Handler Adapter is Simple Controller Handler Adapter (represented by **org.springframework.web.servlet. SimpleControllerHandler Adapter**), which will do the job of the Forwarding the Request from the Dispatcher to the Controller object. Other type of adapter is Throwaway Controller.

88. What interceptors are available in Spring MVC?

Spring MVC allows to intercept web request through handler interceptors. Interception is especially suited for actions such as logging, security, and auditing. The handler interceptor has to implement the HandlerInterceptor interface, which contains three methods:



1. preHandle() – Called before the handler execution, returns a boolean value, “true” : continue the handler execution chain; “false”, stop the execution chain and return it.
2. postHandle() – Called after the handler execution, allow manipulate the ModelAndView object before render it to view page.
3. afterCompletion() – Called after the complete request has finished. Seldom use, can’t find any use case.

In this tutorial, you will create two handler interceptors to show the use of the HandlerInterceptor.

1. ExecuteTimeInterceptor – Intercept the web request, and log the controller execution time.
2. MaintenanceInterceptor – Intercept the web request, check if the current time is in between the maintenance time, if yes then redirect it to maintenance page.

```
public class ExecuteTimeInterceptor extends HandlerInterceptorAdapter{

        private static final Logger logger = Logger.getLogger(ExecuteTimeInterceptor.class);

        //before the actual handler will be executed
        public boolean preHandle(HttpServletRequest request,
                HttpServletResponse response, Object handler)
            throws Exception {

                long startTime = System.currentTimeMillis();
                request.setAttribute("startTime", startTime);

                return true;
        }
```

```
public class MaintenanceInterceptor extends HandlerInterceptorAdapter{
```

```
        //before the actual handler will be executed
        public boolean preHandle(HttpServletRequest request,
                        HttpServletResponse response, Object handler)
            throws Exception {

                Calendar cal = Calendar.getInstance();
                int hour = cal.get(cal.HOUR_OF_DAY);

                if (hour >= maintenanceStartTime && hour <= maintenanceEndTime) {
                        //maintenance time, send to maintenance page
                        response.sendRedirect(maintenanceMapping);
                        return false;
                } else {
                        return true;
                }

        }
}
```

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
     <props>
        <prop key="/welcome.htm">welcomeController</prop>
     </props>
   </property>
   <property name="interceptors">
     <list>
             <ref bean="maintenanceInterceptor" />
             <ref bean="executeTimeInterceptor" />
     </list>
   </property>
</bean>
```

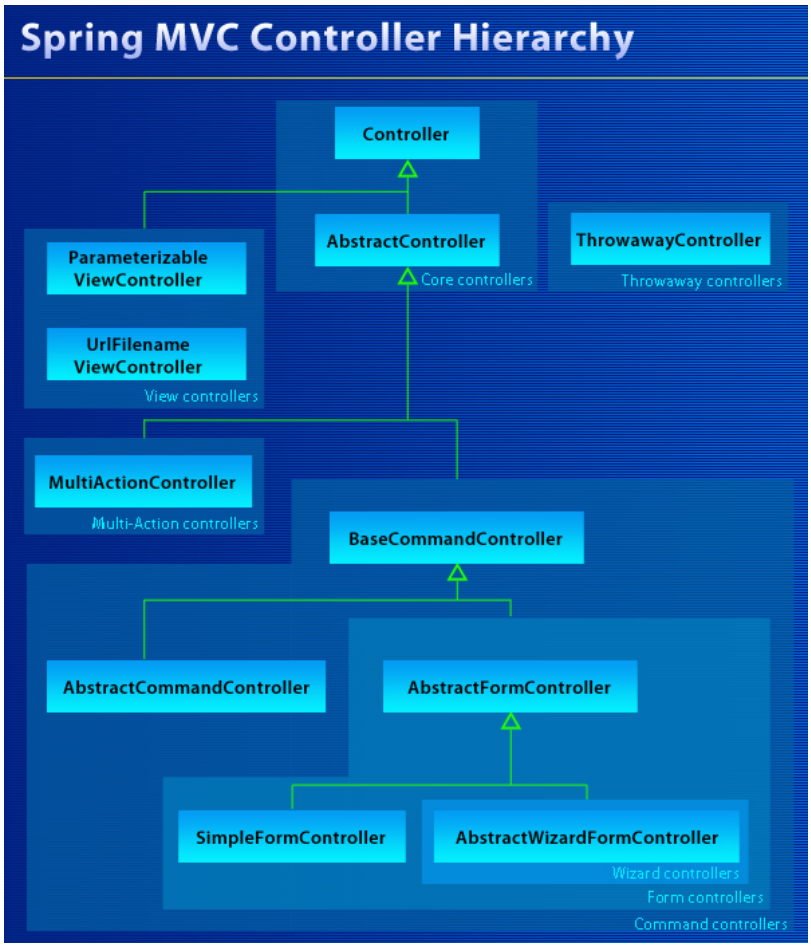
```
<bean id="executeTimeInterceptor"
     class="com.mkyong.common.interceptor.ExecuteTimeInterceptor" />

<bean id="maintenanceInterceptor"
     class="com.mkyong.common.interceptor.MaintenanceInterceptor" >
     <property name="maintenanceStartTime" value="23"/>
     <property name="maintenanceEndTime" value="24"/>
     <property name="maintenanceMapping" value="/SpringMVC/maintenance.htm"/>
</bean>
```

89. How many types of interceptors provided by spring MVC

There are five types of interceptors provided by spring.

1. WebContentInterceptor
2. UserRoleAuthorizationInterceptor
3. OpenSessionInViewInterceptor
4. OpenPersistenceManagerInViewInterceptor
5. LocaleChangeInterceptor

90. What are the different type of Controller available in Spring MVC



91. Define AbstractController?

    The AbstractController is one of the most important abstract base controller providing basic features such as the generation of caching headers and the enabling or disabling of supported methods (GET/POST).

    Workflow

    I. Inspection of supported methods (ServletException if request method is not support)
    II. If session is required, try to get it (ServletException if not found)
    III. Set caching headers if needed according to cacheSeconds propery
    IV. Call abstract method handleRequestInternal() (optionally synchronizing around the call on the HttpSession), which should be implemented by extending classes to provide actual functionality to return ModelAndView objects.

92. Explain ParameterzedViewController

    Trivial controller that always returns a named view. The view can be configured using an exposed configuration property. This controller offers an alternative to sending a request straight to a view such as a JSP.

Workflow

1. Request is received by the controller
2. call to handleRequestInternal which just returns the view, named by the configuration property viewName. Nothing more, nothing less

```
<bean name="parameterizableController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController">
      <property name="viewName" value="ParameterizableController" />
</bean>
```

93. Define MultiActionController

With the help of MultiActionController we can directly call any method inside of the controller. To do this we need to define the MethodNameResolver. Based on the incoming HTTP request, the resolver will decide what method of the MultiActionController will take care of the actual processing and creation of a ModelAndView.

```
<bean class="ticketReservationController"
 class="org.springframework.sample.web.TicketController">
 <property name="methodNameResolver">
  <ref local="reservationActions"/>
 </property>
</bean>

<bean id="reservationActions"  class="org.springframework.web.servlet.mvc...
                              multiaction.PropertiesMethodNameResolver">
 <property name="mappings">
  <props>
   <prop key="/reserve/*.ticket">reserve</prop>
   <prop key="/cancel/*.ticket">cancel</prop>
  </props>
```

94. Explain BaseCommandController

This controller is the base for all controllers wishing to populate JavaBeans based on request parameters, validate the content of such JavaBeans using Validators and use custom editors (in the form of PropertyEditors) to transform objects into strings and vice versa, for example. Three notions are mentioned here:

Command class:

An instance of the command class will be created for each request and populated with request parameters. A command class can basically be any Java class; the only requirement is a no-arg constructor. The command class should preferably be a JavaBean in order to be able to populate bean properties with request parameters.

Populating using request parameters and PropertyEditors:

When a request parameter named 'firstName' exists, the framework will attempt to call setFirstName([value]) passing the value of the parameter. Nested properties are of course supported. For instance a parameter named 'address.city' will result in a getAddress().setCity([value]) call on the command class.

Validators

After the controller has successfully populated the command object with parameters from the request, it will use any configured validators to validate the object. Validation results will be put in a Errors object which can be used in a View to render any input problems.

95. Explain AbstractCommandController

Abstract base class for custom command controllers.Autopopulates a command bean from the request. For command validation, a validator (property inherited from BaseCommandController) can be used.In most cases this command controller should not be used to handle form submission, because functionality for forms is offered in more detail by the AbstractFormController and its corresponding implementations.

96. Explain AbstractFormController?

AbstractFormController, auto-populates a form bean from the request. This, either using a new bean instance per request, or using the same bean when the sessionForm property has been set to true.This class is the base class for both framework subclasses such as SimpleFormController and AbstractWizardFormController and custom form controllers that you may provide yourself.

A form-input view and an after-submission view have to be provided programmatically. To provide those views using configuration properties, use the SimpleFormController.`

Subclasses need to override showForm to prepare the form view, and processFormSubmission to handle submit requests. For the latter, binding errors like type mismatches will be reported via the given "errors" holder. For additional custom form validation, a validator (property inherited from BaseCommandController) can be used, reporting via the same "errors" instance.

Workflow

1. The controller receives a request for a new form (typically a GET).
2. Call to formBackingObject() which by default, returns an instance of the commandClass that has been configured.
3. Call to initBinder() which allows you to register custom editors for certain fields (often properties of non-primitive or non-String types) of the command class.
4. Only if bindOnNewForm is set to true, then ServletRequestDataBinder gets applied to populate the new form object with initial request parameters and the onBindOnNewForm (HttpServletRequest, Object, BindException) callback method is called.
5. Call to showForm() to return a View that should be rendered (typically the view that renders the form). This method has to be implemented in subclasses.

6. The showForm() implementation will call referenceData(), which you can implement to provide any relevant reference data you might need when editing a form.
7. Model gets exposed and view gets rendered, to let the user fill in the form.
8. The controller receives a form submission (typically a POST). To use a different way of detecting a form submission, override the isFormSubmission method.
9. If sessionForm is not set, formBackingObject() is called to retrieve a form object. Otherwise, the controller tries to find the command object which is already bound in the session. If it cannot find the object, it does a call to handleInvalidSubmit which - by default - tries to create a new form object and resubmit the form.
10. The ServletRequestDataBinder gets applied to populate the form object with current request parameters.
11. Call to onBind(HttpServletRequest, Object, Errors) which allows you to do custom processing after binding but before validation
12. If validateOnBinding is set, a registered Validator will be invoked. The Validator will check the form object properties, and register corresponding errors via the given Errors
13. object.
14. Call to onBindAndValidate() which allows you to do custom processing after binding and validation (e.g. to manually bind request parameters, and to validate them outside a Validator).
15. Call processFormSubmission() to process the submission, with or without binding errors. This method has to be implemented in subclasses.

97. Explain SimpleFormController

Concrete FormController implementation that provides configurable form and success views, and an onSubmit chain for convenient overriding. Automatically resubmits to the form view in case of validation errors, and renders the success view in case of a valid submission.

Workflow

1. The controller receives a request for a new form (typically a GET).
2. Call to formBackingObject() which by default, returns an instance of the commandClass that has been configured.
3. Call to initBinder() which allows you to register custom editors for certain fields (often properties of non-primitive or non-String types) of the command class.
4. Only if bindOnNewForm is set to true, then ServletRequestDataBinder gets applied to populate the new form object with initial request parameters and the onBindOnNewForm(HttpServletRequest, Object, BindException) callback method is called.
5. Call to showForm() to return a View that should be rendered (typically the view that renders the form). This method has to be implemented in subclasses.
6. The showForm() implementation will call referenceData(), which you can implement to provide any relevant reference data you might need when editing a form.
7. Model gets exposed and view gets rendered, to let the user fill in the form.
8. Call to processFormSubmission which inspects the Errors object to see if any errors have occurred during binding and validation.
9. If errors occured, the controller will return the configured formView, showing the form again (possibly rendering according error messages).
10. If isFormChangeRequest is overridden and returns true for the given request, the controller will return the formView too. In that case, the controller will also suppress validation. Before returning the formView, the controller will invoke onFormChange(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, java.lang.Object, org.springframework.validation.BindException), giving sub-classes a chance to make modification

to the command object. This is intended for requests that change the structure of the form, which should not cause validation and show the form in any case.

11. If no errors occurred, the controller will call onSubmit using all parameters, which in case of the default implementation delegates to onSubmit with just the command object. The default implementation of the latter method will return the configured successView. Consider implementing doSubmitAction(java.lang.Object) doSubmitAction for simply performing a submit action and rendering the success view.

98. What are the different types of View resolver available in Spring MVC

After completion of request processing by a controller, we need to redirect it to a view. Controller gives a view name and DispatcherServlet maps the view name to an appropriate view based on the configuration provided and redirects/forwards the request to that view. Spring MVC provides following types of view resolvers. We can have multiple view resolvers chained in the configuration files.

- AbstractCachingViewResolver: Extending this resolver provides ability to cache the views before actually calling the views.
- XMLViewResolver: Takes view configuration in xml format compliant with DTD of Spring's bean factory. Default configuration is searched in WEB-INF/views.xml.
- ResourceBundleViewResolver: Definitions are searched in resource bundle i.e. property files. Default classpath property file is searched with name views.properties.
- UrlBasedViewResolver: Straightforward url symbol mapping to view.
- InternalResourceViewResolver: Subclass of UrlBasedViewResolver that supports JSTL and Tiles view resolving.
- VelocityViewReolver: Subclass of UrlBasedViewResolver used to resolve velocity views.
- FreeMarkerViewResolver: Subclass of UrlBasedViewResolver used to resolve FreeMarker views.