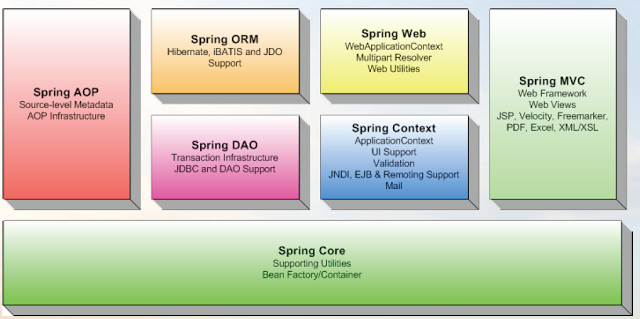
[Spring Core Concepts](http://www.trvajjala.in/2010/12/spring-core-concepts.html)

**1. Spring Modules**  
  
The Spring framework is a layered architecture consisting of seven well-defined modules. The Spring modules are built on top of the core container, which defines how beans are created, configured, and managed, as shown in following figure. 

[](http://4.bp.blogspot.com/_L1yyh0JUG7Q/TS6HTDXPMaI/AAAAAAAAAd8/zAEOcFyDbjw/s1600/SpringFrameworkModules.PNG)

Each of the modules or components that consist of the Spring framework can stand on its own or be implemented jointly with one or more of the other modules or components. The description of each component is as follows:

1. **The core container**:

The core container provides the fundamental functionality of the spring framework. In this module primary component is the BeanFactory, an implementation of the Factory pattern. The BeanFactory applies the Inversion of Control (IOC) pattern to separate an application's configuration and dependency specification from the actual application code.

1. **Spring context module**:

The Spring context is a configuration file that provides context information to the Spring framework. The spring context includes enterprise services such as e-mail, JNDI, EJB, internalization, validation, scheduling and applications lifecycle events. Also included is support for the integration with templating frameworks such as velocity.

1. **Spring AOP module**:

The Spring AOP module allows a software component to be decorated with additional behavior, through its configuration management feature. As a result you can easily AOP-enable any object managed by the Spring framework. The Spring AOP module provides transaction management services for objects in any Spring-based application. With Spring AOP you can incorporate declarative transaction management into your applications without relying on EJB components.

1. **Spring DAO module**:

The Spring DAO module provides a JDBC-abstraction layer that reduces the need to do tedious JDBC coding and parsing of database-vendor specific error codes. Also, the JDBC package provides a way to do programmatic as well as declarative transaction management, not only for classes implementing special interfaces, but for all your POJOs (plain old Java objects).

1. **Spring ORM module**:

 Spring provides integration with OR mapping tools like Hibernate, JDO and iBATIS. Spring transaction management supports each of these ORM frameworks as well as JDBC.

1. **Spring Web module**:

The Web context module provides basic web-oriented integration features builds on top of the application context module, providing contexts for Web-based applications. As a result, the Spring framework supports integration with Jakarta Struts. The Web module also eases the tasks of handling multi-part requests and binding request parameters to domain objects.

1. **Spring MVC framework module**:

Spring provides a pluggable MVC architecture. The users have a choice to use the web framework or continue to use their existing web framework. Spring separates the roles of the controller; the model object, the dispatcher and the handler object which makes it easier to customize them. Spring web framework is view agnostic and does not push the user to use only JSPs for the view. The user has the flexibility to use JSPs, XSLT, velocity templates etc to provide the view.

**2. Basic Bean Writing:**

       Spring based applications objects will live within the spring container.

       The container will create the objects, wire them together, configure them, and manage their complete lifecycle from cradle to grave (means **new** to finalize ())

       Container is the place the objects are hanging out.

       Spring’s container uses dependency injection to manage the components that make up an application. This includes creating associations between collaborating components.

       These Spring Objects are cleaner and easy to understand, support reuse, and are easy to **unit-test.**

       **spring comes with Several Container implementations**(means is not having only one container, it is more than one). **That can be categorized into two distinct types.**

**a.     Bean factories**are the simplest container, providing basic support for Dependency Injection.

**b.     Application Contexts**build on notion of a bean factory by providing application framework services, such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

**2.1 BeanFactory Type Containers**

       Bean factory is an implementation of the *Factory Design Pattern*. It is a class whose responsibility is to create and dispense beans of many types.

       Bean factory takes part in the lifecycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.

       BeanFactory implementation containers must implement the interface***org.springframework.beans.factory.BeanFactory***

       There are several implementations of BeanFactory in spring. But the one most commonly used is*org.springframework.beans.factory.xml.XmlBeanFactory,*which loads its beans based on the definitions contained in an XML file.

       To create **XmlBeanFactory**, we must pass an instance of *org.springframework.core.io.Resource*to the Constructor.

   Resource resource=new FileSystemResource ("c:/mybeans.xml");

         XmlBeanFactory xbf=new XmlBeanFactory (resource);

       Beans are “lazily” loaded into bean factories, meaning that beans are instantiated when they are required.(useful in mobile applications)

       To retrive a bean from a BeanFactory, simply call the getBean () method, passing the ID of the bean you want to retrive.

   MyBean myBean =(MyBean) xbf.getBean("myBean");

       There are total **Eight** resources available in the spring which implements *org.springframework.core.io.Resource*interface. In this **six** are in *org.springframework.core.io.\** package and other **two** are in*org.springframework.web.portlet.context.PortletContextResource , org.springframework.web.context.support.ServletContextResource*

       Six resources are defined below as

*org.springframework.core.io.ByteArrayResource;*

*org.springframework.core.io.ClassPathResource;*

*org.springframework.core.io.FileSystemResource;*

*org.springframework.core.io.InputStreamResource;*

*org.springframework.core.io.UrlResource;*

*org.springframework.core.io.DescriptiveResource;*

**2.2 Application Context Type Containers**

       This is spring’s more advanced container, which offers more services such as

                                i.            Resolving and loading text messages from properties file.

                               ii.            Supports for internationalization (i18n) of above messages.

                             iii.            It provides generic way to load file resources, such as images.

                            iv.            It can able to publish events to beans that are registered as listeners.

       Among the many implementations of ApplicationContext are three that are commonly used:

         **ClassPathXmlApplicationContext**- Loads a Context definition from an xml file located in the classpath, treating context definition files as classpath resources.

         **FileSystemXmlApplicationContext**-Loads a context definition from an XML files in the file system.

         **XmlWebApplicationContext**-Loads context definitions from an XML file contained within a web application.

       Major difference between ApplicationContext and BeanFactory is how singleton beans are loaded. A **BeanFactory**lazily loads all beans, deferring (postpone, suspend) bean creation until the getBean () method is called.

**ApplicationContext** is a bit smarter and preloads all singleton beans upon context startup.

       **ApplicationContext** interface extends **BeanFactory** interface

**2.3 Bean’s Life Cycle**

       BeanFactory containers beans life cycle methods

1.      Instantiate

2.      Populates properties

3.      Set beanName

4.      Set BeanFactory

5.      PostProcess

6.      Initialize beans

7.      Postprocess

8.      Bean ready to use

9.      Destroy bean

       ApplicationContext container beans life cycle methods

1.      Instantiate

2.      Populates properties

3.      Set beanName

4.      Set BeanFactory

5.      Set ApplicationContext

6.      PostProcess

7.      Initialize beans

8.      Postprocess

9.      Bean ready to use

10.  Destroy bean

**2.4 Creating Beans**

       Injecting through the constructor

i)   <bean id="one" class="com.one.One">

         <constructor-arg value="SomeText...."/>

     </bean>

           ii)   <bean id="one" class="com.one.One">

          <constructor-arg>

            <bean class="com.Test"/>

        </constructor-arg>

     </bean>

       Injecting through the property

i) <bean id="one" class="com.one.One">

        <property name="name" value="injecting through Property"/>

  </bean>

ii) <bean id="one" class="com.one.One">

        <property name="test” ref=”test” />

   </bean>

        Wiring Collections

i)    <bean id="one" class="com.one.One">

        <property name="colleges">

            <set>

                <value>One</value>

                <value>One</value>

                <value>Two</value>

            </set>

        </property>

    </bean>

ii)    <bean id="one" class="com.one.One">

        <property name="colleges">

            <list>

                <value>One</value>

                <value>One</value>

                <value>Two</value>

            </list>

        </property>

    </bean>

iii)    <bean id="one" class="com.one.One">

        <property name="colleges">

            <map>

                        <entry key=”” value=””/>

                        <entry key-ref=”” value-ref=””/>

            </map>

        </property>

    </bean>

iv)  <bean id="one" class="com.one.One">

        <property name="colleges">

             <props>

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

                <prop key="one" **value**=”dd”/> is wrong

            </props>

        </property>

    </bean>

 v) <bean id="one" class="com.one.One">

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

    </bean>

**2.5 Auto Wiring**

      Spring’s another feature which enable automatically *figure out* how to wire beans together by setting the autowire property on each bean that we want.

      Spring supports four types of autowiring

o        ByName

    <bean id="one" class="com.one.One" autowire="byName">

    </bean>

o        ByType

    <bean id="one" class="com.one.One" autowire="byType">

    </bean>

o        Constructor

    <bean id="one" class="com.one.One" autowire="constructor">

    </bean>

o        Autodetect

    <bean id="one" class="com.one.One" autowire="autodetect">

    </bean>

      If bean has been configured to autowire by ***autodetect****,*spring will attempt to autowire by constructor first, if no such constructor-bean found it will attempt to autowire by type.

      Instead of writing autowire on each bean we can set default-autowire="" on beans element as explicit wiring

<beans default-autowire

="autodetect|constructor|byType|byName">

…….

</beans>

      **Autowire** dominates the **default-autowire** wiring

      **Drawbacks of auto-wiring**(work: list out based on above criteria)

**2.6 Controlling Bean Creation**

       Controlling how many instances of a specific bean are created, whether it is one instance for the entire application(singleton), one instance per user request(request), or a brand-new instance each time the bean is used(prototype).

       Create beans from static factory methods instead of public constructors.

       Initialize bean after it is created and clean up just before it is destroyed.

       Spring is five different types of scopes

o        singleton: single instance per container

o        prototype: new instance every time

o        request: valid in Spring MVC

o        session: valid in Spring MVC

o        global-session :valid in a portlet context

       **Initialization on demand holder** (design pattern): it is a lazy loaded singleton.the idiom can be implemented in both single-threaded/serial and concurrent environments, but care must be taken to correctly implement this.

public class Stage {

    private Stage() {

    }

    private static class StageSingletonHolder {

        static Stage INSTANCE = new Stage();

    }

    public static Stage getInstance() {

        return StageSingletonHolder.INSTANCE;

    }

}

       Calling static methods using factory method

<bean id="stage" class="com.Stage" factory-method="getInstance">

            </bean>

**2.7 Initializing and destroying beans**

This can be possible in three ways

1) By writing our own custom initializing and destroy methods that we need to mention below as follows.

     <bean id=”test” class=”com.Test”

                            init-method=”myInit”

                            destroy-method=”myDestroy”>

     </bean>

2) By declaring  globally for all beans common init and destroy methods

     <beans default-init-method=”myInit” default-destroy-method=”myDestroy”>

            ……

     </beans>

3)By implementing the interfaces

org.springframework.beans.factory.InitializingBean

org.springframework.beans.factory.DisposableBean

spring container automatically identifies and calls **afterPropertiesSet()** and  **destroy()**methods.

**2.8 Working with session and request scopes**

a)If you are accessing scoped beans within Spring Web MVC, i.e. within a request that is processed by the SpringDispatcherServlet, or DispatcherPortlet, then no special setup is necessary: DispatcherServlet and DispatcherPortlet already expose all relevant state.

b) When using a Servlet 2.4+ web container, with requests processed outside of Spring's DispatcherServlet (e.g. when using JSF or Struts), you need to add the following javax.servlet.ServletRequestListener to the declarations in your web application's'web.xml' file.

<listener>

    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>

 </listener>

c) If you are using an older web container (Servlet 2.3), you will need to use the provided javax.servlet.Filter implementation. Find below a snippet of XML configuration that has to be included in the 'web.xml' file of your web application if you want to have access to web-scoped beans in requests outside of Spring's DispatcherServlet on a Servlet 2.3 container. (The filter mapping depends on the surrounding web application configuration and so you will have to change it as appropriate.)

<web-app>

  ..

  <filter>

    <filter-name>requestContextFilter</filter-name>

    <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>

  </filter>

  <filter-mapping>

    <filter-name>requestContextFilter</filter-name>

    <url-pattern>/\*</url-pattern>

  </filter-mapping>

  ...

</web-app>

d) we can use like this also

<listerner>

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

</listener>

Here we need to tell the location of the Spring Configuration files also. By default it will look for file at /WEB-INF/applicationContext.xml .  but we can change this by mentioning like below.

<context-param>

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

<param-value>

/WEB-INF/moduleOne/one.xml

/WEB-INF/moduleTwo/two.xml

/WEB-INF/moduleThree/three.xml

</param-value>

</context-param>

c)  Some web container do not initialize servlet listeners before servlets . this case we need to use **ContextLoaderServlet.**

**3.Advance Bean Wiring**

**3.1 Declaring Parent and Child beans**

Inorder to reduce the redundancy in the spring configuration files <bean> element provides two special attributes

1. parent : it is similar to *extends* in java Class
2. abstract : this never instantiated by the Spring

**3.1.1 Abstracting base bean type**

         <bean id=”one” class=”com.One” **abstract**=”true”/>

<bean id=”two” **parent**=”one”/>

<bean id=”three” **parent**=”one”/>

**3.1.2 Abstracting common properties**

        <bean id=”myproperty”  **abstract**=”true”>

            <property name=”song” value=”sssssssss”/>

         </bean>

<bean id=”two” **parent**=”myproperty”/>

<bean id=”three” **parent**=” myproperty”/>

**3.2 Applyting method injection**

Setter injection allows to inject values into a bean’s properties.method injection allows to inject entire method definitions into a bean.

Spring supports two forms of method injection

1. Method replacement
2. Getter Injection

**3.2.1 Basic method replacement**

**3.2.2 Using getter injection**

**3.3 Injecting non-spring beans**

**3.4 Registering custom property editors**

**3.5 Working with Spring Special beans**

**3.5.1 Postprocessing the Beans**

**3.5.2 PostProcessing the BeanFactory**

**3.5.3 Externalize configuration properties**

**3.5.4 Resolving text messages**

**3.5.5 Decoupling with application events**

**3.5.6 Making Bean Aware**

**3.6 Scripting Beans With Groovy**

1. **Advising Beans**

In software development, functions that span multiple points of an application are called *cross-cutting concerns*. Typically these cross-cutting concerns are conceptually separated from the application’s business logic. Separating these cross-cutting concerns from business logic is where AOP goes to work.

       DI decuples application objects from each other.

       AOP decuples cross-cutting concerns from the objects that they affect.

Any Object-Oriented technique is mainly focuses on re-using functionality that can be achieved by following ways

1)     Inheritance

2)     Delegation

3)     Aspects

**4.1 AOP terminology:**

**a)     cross-cutting concerns:** any functionality that affects multiple points of an application.

**b)     Adive**

**c)     JointPoint**

**d)     PointCut**

**e)     Aspect**

**f)      Introduction**

**g)     Target**

**h)     Proxy**

**i)       Weaving**

       Spring’s support for AOP comes in four flavors:

1)     Classic Spring proxy-based AOP

2)     @AspectJ annotation-driven aspects

3)     Pure-POJO aspects

4)     Injected AspectJ aspects

       Spring **advice** are written in Standard Java Class and **pointcuts** are written in xml configuration files.

       Spring advices objects at runtime: aspects are woven into spring-managed beans at runtime by wrapping them with a proxy class.

       **Spring generates proxy classes in two ways.**

o        using **java.lang.reflect.Proxy**

o        using **CGLIB**(Code Generation Library) to generate subclass to the target class at runtime.

       Methods marked as *final* cannot be advised because CGLIB can’t generate subclass to it.

       AOP frameworks supports three types of jointpoints

o        Field jointpoints

o        Constructor jointpoints

o        Method jointpoints

       Spring supports only method jointpoints, we cannot apply advice when bean is instantiated because sping doesn’t supports constructor jointpoints. We can’t apply advice when field is updated in a object because Spring doesn’t support field jointpoints.

       In spring ***five types of Advice*** implementation and ***two types of Pointcut*** implementations available.

       **AOP Alliance** is an open source project whose goal is to facilitate and standardize AOP.

       Except **MethodInterceptor**  all interfaces are comes with Springframework.

**4.2 Defining Advice**

|  |  |
| --- | --- |
| **Advice Type** | **Interface** |
| Before | org.springframework.aop.MethodBeforeAdvice |
| After-returning | org.springframework.aop.AfterReturningAdice |
| After-throwing | org.springframework.aop.ThrowsAdvice |
| Introduction | org.springframework.aop.IntroductionInterceptor |
| Around | org.aopalliance.intercept.MethodInterceptor |

       Before advice need to implement **MethodBeforeAdice** interface and implement  **before()** method

  public void **before**(Method method,Object[] args,Object target) throws Throwable;

                        method takes three parametes

a)      java.lang.reflect.Method : this represents method to which the advice is being applied.

b)      Object[]: arguments to the method

c)       Object : object on which the method was called.

       After-returning advice need to implement **AfterReturningAdvice** interface and implement afterReturning() method

public void **afterReturning**(Object returnValue,Method method,Object[],Object obj) throws Throwable;

       After-throwing advice need to implement **ThrowsAdvice** interface

public void afterThrowing([method],[args],[target],throwable);

all parameters are optional except **throwable**one

(ThrowsAdvice is marker interface but it checks for **afterThrowing** signature method)

       Around Advice need to implement **MethodInterceptor**interface and works before and after returning and afterthrowing cases.

public Object **invoke**(MethodInvocation invocation) throws Throwable;

       sample implementation how it satisfies above three Conditions

     public Object **invoke**(MethodInvocation **invocation**) throws Throwable{

            try{

                        --------------executed MethodBeforeAdvice---------------

**invocation.proceed();**

                        --------------executed AfterReturningAdvice---------------

            }catch(Exception e){

                        --------------executed   throwsAdvice---------------

            }

     }

       Failure to call **proceed()** will result in the advice being applied but the target method never being executed.

**4.3 Defining Pointcuts**

       Spring comes with several different types of Pointcuts to choose from,but two are the most useful pointcuts.

|  |  |
| --- | --- |
| **Pointcut Type** | **Java Class** |
| RegularExpressionPointcut | org.springframework.aop.support.JdkRegexpMethodPointcut |
| AspectJExpressionPointcut | org.springframework.aop.aspectj.AspectJExpressionPointcut |

       Spring comes with two classes that implement **RegularExpressionPointcuts**

1)          org.springframework.aop.support.Perl5RegexpMethodPointcut– useful when application will be running in a pre-java1.4 environment.(it Required Jakarta ORO).

2)          org.springframework.aop.support.JdkRegexpMethodPointcut-best choice if we run jdk1.4 or higher.

       **AspectJExpressionPointcuts** are defined by using the  below class

org.springframework.aop.aspectj.AspectJExpressionPointcut

**4.4 Defining Advisors**

       After defining Advice and pointcuts  *we need to associate this pointcut with advice.*Spring will do this  with Three types of **Advisors**

1)          org.springframework.aop.support.DefaultPointcutAdvisor

2)          org.springframework.aop.support.RegexpMethodPointcutAdvisor

3)          org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor

      All above Advisors extends   one  abstract class org.springframework.aop.support.**AbstractGenericPointcutAdvisor**

**4.5 Example to use Aspects**  
**Example  to Create Advice**

<bean id=”myAdvice” class=”com.MyAdvice”></bean>

**Example to Create RegularExpress Pointcut**

<bean id=”regexPointcut” class=” org.springframework.aop.support.JdkRegexpMethodPointcut”>       <property name=”pattern” value=”.\*perform” ”/>

 </bean>

**Example to Create AspectJ Pointcut**

  <bean id=”aspectJPointcut” class=” org.springframework.aop.aspectj.AspectJExpressionPointcut”>

<property name=”expression” value=”execution(\*  \*.perform”/>

   </bean>

**Example to Use DefaultPointcutAdvisor**

<bean id=”defaultAdvisor” class=” org.springframework.aop.support.DefaultPointcutAdvisor”>

<property name=”advice” ref=” myAdvice”/>

<property name=”pointcut” ref=” regexPointcut”/>

<property name=”pointcut” ref=” aspectJPointcut”/>  <!--   any one 

     </bean>

**Example to Use RegexpMethodPointcutAdvisor**

<bean id=”regexAdvisor” class=” org.springframework.aop.support.RegexpMethodPointcutAdvisor”>

<property name=”advice” ref=” myAdvice”/>

<property name=”pattern”  value=”.\*perform”/>

     </bean>

**Example to Use AspectJExpressionPointcutAdvisor**

    <bean id=”aAdvisor” class=” org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor”>

<property name=”advice” ref=” myAdvice”/>

<property name=”expression”  value=”execution(\* .\*perform (..))”/>

     </bean>

**4.6 Using Proxy Factory Beans**

In Spring AOP, *advisors* completely define as *aspect* by asscociating *advice* with a *pointcut*. But Aspects in Spring are proxied. Whether you use Regular expression pointcuts or AspectJ pointcuts,you still need to proxy your target beans for the advisors to take effect. For that we need to declare one or more **ProxyFactoryBeans.**

            <bean id=”duke” class=”org.springframework.aop.framework.ProxyFactoryBean”>

               <property name=”target” ref=”dukeTarget”/>

              <property name=”interceptorNames” value=”someAdvisor”/>

              <property name=”proxyInterfaces” value=”com.SomeInterFace”/>

            </bean>

But it requires to write more xml in the configuration file hence I’m not going into deep into this.

**4.7 AutoProxying**

There are **two** ways to autoproxy beans

*Basic autoproxying of beans based on advisor beans declared in the Spring context*-The Advisor’s pointcut expression is used to determine which beans and which methods will be proxied.

*Autoproxying based on @AspectJ annotation-driven aspects-*The pointcuts pecified on the advice contained within the aspect will be used to choose which beans and methods will be proxied.

**First Method Implementation**(Creating autoproxies for Spring Aspects): After creating Advisor as shown in the **Example to Use AspectJExpressionPointcutAdvisor**part . we need to add some bean configuration into the xml file.

<bean class=”org.springframework.aop,framework,autoproxy.DefaultAdvisorAutoProxyCreator”/>

**NOTE: DefaultAdvisorAutoProxyCreator** is the implementation of **BeanPostProcessor**interface ,which recognized by Spring Container and creates proxies runtime.

**Second Method Implementation**(Autoproxying **@AspectJ** aspects): using this we need to add some annotations to the class as shown below.

       Annotation **Around** advice implementation will have only one method watchPerformance() which takes**ProceedingJointPoint** as argument

**@Around**(“performance()”)

public void watchPerformance(ProceedJointPoint **jointpoint**){

try{

**jointpoint.proceed();**

}catch(Exception e){

}

}

Add <aop:aspectj-autoproxy/> to the configuration file inorder to work below code.

**@Aspect**

public class Audience{

**@Pointcut**(“execution(\* \*.perform(..))”)

  public void performance(){}

**@Before**(“performance()”)

  public void takeSeats(){

  System.out.println(“do something ”);

 }

**@AfterReturning**(“performance()”)

  public void appalaud(){

 System.out.println(“do something ”);

  }

**@AfterThrowing**(“performance()”)

 public void demandRefund(){

  System.out.println(“do something ”);

}

}

**4.8 Declaring Pure-POJO aspects**

Using This we need not required Source code on which we apply Aspects. Declaration of this kind of aspects as shown below.

<beans>

             <bean id=”audience” class=”com.Audience”/>

<aop:config>

                <aop:aspect ref=”audience”>

                  <aop:pointcut id=”performance” expression=”execution(\* \*.perform(..))”/>

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

     </aop:aspect>

</aop:config>

</beans>

**4.9 Injecting AspectJ aspects**

This is totally AspectJ Framework through which we can Advice Constructor and fields also ,this is not part of spring framework hence I’m not discussing here. 

**5 HITTING DATEBASE**

**6 MANAGING TRANSACTIONS**

**Definition :** Transaction is a series of sql statements that either succeeds or fails as a unit to prevent inconsistent data.

Transaction should pass ACID Test.

***Atomic***: It ensures that all the operations in the transaction happen or that none of them happen.

***Consistent***: Once the transaction ends (success or fail) the system is left in a state that is consistent.

***Isolated***: transaction should be isolated from each other,preventing concurrent reads and writes to the same data (involves locking rows and tables in database).

***Durable***: once the transaction completed the results should be permanent.

|  |  |  |
| --- | --- | --- |
| **Type** | **EJB** | **Spring** |
| Programmatic | JTA | Callback Mechanism |
| Declarative | CMT | Proxing Beans, XML-Declared, annotation-driven transactions. |

Spring doesn’t directly manage transactions. Instead ,it comes with a selection of transaction managers that delegates responsibilities for transaction management to a platform-specific transaction implementation provided by either JTA or the persistent mechanism.

1. **JDBC transactions**

<bean id=”transactionManager”         class=”org.springframework.jdbc.datasource.DataSourceTransactionManager”>

<property name=”dataSource” ref=”dataSource”/>

</bean>

1. **Hibernate transactions**

<bean id=”transactionManager”         class=”org.springframework.orm.hibernate3.HibernateTransactionManager”>

<property name=”sessionFactory” ref=” sessionFactory”/>

</bean>

**6.3 Programmatic Transactions in Spring**

Step 1: Inject transactionTemplate into the Service/Dao Class.

<bean id="testService" class="com.trx.TestServiceImpl">

        <property name="transactionTemplate">

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

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

            </bean>

        </property>

</bean>

Step2 :  write code as shown below in the java class

public void addBook(Book book) {

        transactionTemplate.execute(new TransactionCallback() {

            public Object doInTransaction(TransactionStatus ts) {

                try {

                } catch (Exception e) {

                    ts.setRollbackOnly();

                }

                return null;

            }

        });

    }

**6.4             Declarative Transactions**

Spring offers declarative transactions to POJOs. Spring declarative transaction management is implemented through Spring’s AOP framework.

Spring Three ways Supports Declarative Transactions :

1)     By Proxying beans using Spring AOP.

2)     XML-Declared transactions.

3)     Annotation-driven transactions.

**Transaction Attributes** :  Transaction attribute is a description of how transaction policies should be applied to a method.

       I.      propagation

     II.      isolation

  III.      rollback rules

   IV.      timeout

     V.      read-only?

**Propagation Behaviour :** Propagation behaviour defines the boundaries of transaction with respect to the client and to the method being called.

Note : *org.springframework.transaction.TransactionDefinition* is an interface which defines the below constants.

|  |  |
| --- | --- |
| **Propagation Behavior** | **Description** |
| PROPAGATION\_MANDATORY | Method must run with in transaction . otherwise an exception will be thrown. |
| PROPAGATION\_NESTED | Method should run within anested transaction if an existing transaction is in progress. |
| PROPAGATION\_NEVER | Method should not run within transaction. otherwise an exception will be thrown. |
| PROPAGATION\_NOT\_SUPPORTED | Method should not run within a transaction. If there transaction exists that suspended. |
| PROPAGATION\_REQUIRED | Method must run within transaction . if no transaction exists it will create. |
| PROPAGATION\_REQUIRES\_NEW | Method must run within its own transaction. A new transaction is started and if existing one is in progress. |
| PROPAGATION\_SUPPORTS | Method doesnot requires transactional context, but may run within a transaction if one is already in progress. |

**Isolation Levels :**it defines how much a transaction may be impacted by the activities of other concurrent transactions. In a typical application, multiple transactions run concurrently,often working with the same data to get their job done. It leads to the following problems.

       **Dirty read:**it occurs one transaction reads the data that has been written (but not committed ) by  the another transaction. If the changes are rolled back by the second transaction , the data obtained by the first transaction will be invalid.

       **Nonrepeatable read:** if the same transaction performs read query two or more times and each time the data is different , this is because another concurrent transaction updating the data between the queries.

       **Phantom reads:**similar to the nonrepeatable read. (??)

In the ideal case ,transactions would be completely isolated from each other. Perfect isolation can affect  performance because it often involves locking rows in the data store.

|  |  |
| --- | --- |
| **Isolation Level** | **Description** |
| ISOLATION\_DEFAULT |  |
| ISOLATION\_READ\_UNCOMMITTED |  |
| ISOLATION\_READ\_COMMITTED |  |
| ISOLATION\_REPEATABLE\_READ |  |
| ISOLATION\_SERIALIZABLE |  |

**Read-only :**

**Transaction timeout:**

**Rollback rules:**

**7            SPRING WEB MVC FRAMEWORK**

Spring Web MVC framework is designed around a *DispatchServlet*that dispatches requests to handlers,with configurable handler mappings,view resolution,locale and theme resolution as well as support for upload files.

Note: it is not possible to add advice to final methods using Spring MVC. For example we can’t add advice toAbstractController.handleRequest() method.

       Each DispatchServlet has its own WebApplicationContext,which inherits all the beans already defined in the root WebApplicationContext.

       The framework will, on initialization of a DispatcherServlet, *look for a file named  [servlet-name]-servlet.xml*in theWEB-INF directory of your web application and  create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

       **Special beans in the WebApplicationContext**

a.      Controllers

b.      HandlerMappings

c.      View Resolvers

d.      Locale Resolver

e.      Theme Resolver

f.       Multipart file Resolver

g.      Handler Exception Resolver

Complete process for a request which goes is handled by the DispatchServlet.

**Step 1:** The WebApplicationContext is searched and bound to under key

DispatchServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE.

**Step 2:**locale resolver is bound to the request to let elements in the process resolve the locale to use when processing the request.

**Step 3:** Theme Resolver is bound to the request to let elements such as views determine which theme to use.

**Step 4:**if a multipart resolver is specified ,the request is inspected for multiparts,if multiparts are found,the request is wrapped to MultipartHttpServletRequest.

**Step 5:** An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) will be executed in order to prepare a model (for rendering).

**Step 6:** If a model is returned, the view is rendered. If no model is returned (which could be due to a pre- or postprocessor intercepting the request, for example, for security reasons), no view is rendered, since the request could already have been fulfilled.

**Note 1:** Exceptions that are thrown during processing of the request get picked up by any of the handler exception resolvers that are declared in the WebApplicationContext.

**Note 2:**Spring DispatchServlet also has support for returning the last-modification-date. The DispatcherServlet will first lookup an appropriate handler mapping and test if the handler that is found *implements the interface LastModified*interface. If so, the value of the long getLastModified(request) method of the LastModified interface is returned to the client.

**DispatchServlet**initialization parameters

|  |  |
| --- | --- |
| **Parameter** | **Explanation** |
| contextClass | Class that implements WebApplicationContext, which will be used to instantiate the context used by this servlet. If this parameter isn't specified, the  XmlWebApplicationContext will be used. |
| contextConfigLocation | String which is passed to the context instance(specified by contextClass) to indicate where context(s) can be found. The string is potentially split up into multiple strings(using a comma as delimiter) to support multiple contexts( in case of multiple context locations,of beans that are defined twice,the latest takes precendence. |
| namespace | The namespace of the WebApplicationContext.Defaults to [servlet-name]-servlet. |

**Spring Controllers**

**1. Controller interface :**Spring's basis for the controller architecture is the org.springframework.web.servlet.mvc.Controller which is having one method.

public class SimpleOneController implements Controller{

public ModelAndView **handleRequest**(HttpServletRequest arg0, HttpServletResponse arg1) throws Exception {

            return new ModelAndView("index");

}

}

**2. AbstractController and WebContentGenerator:**

All of Spring’s various Controllers inherit from AbstractController .

This class offering caching support.

public class SimpleTwoController extends AbstractController{

   @Override

protected ModelAndView **handleRequestInternal**(HttpServletRequest arg0, HttpServletResponse arg1)  throws Exception {

        return new ModelAndView("index");

    }

}

Features offered by the **AbstractController**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Default** | **Explanation** |
| supportedMethods | GET,POST | Indicates what methods this controller should accept. |
| requireSession | false | indicates whether or not this controller requires a HTTP session to do its work |
| synchronizeOnSession | false | use this if you want handling by this controller to be synchronized on the user's HTTP session. |
| cacheSeconds | -1 | when you want a controller to generate a caching directive in the HTTP response, specify a positive integer here. By default the value of this property is set to *-1*so no caching directives will be included in the generated response. |
| useExpiresHeader | - | tweaks your controllers to specify the HTTP 1.0 compatible*"Expires"*header in the generated response. By default the value of this property is true. |
| useCacheHeader | - | tweaks your controllers to specify the HTTP 1.1 compatible*"Cache-Control"*  header in the generated response. By default the value of this property is true. |

**3.**The **ParameterizableViewController** is basically the same as the example above, except for the fact that you can specify the view name that it will return in the web application context (and thus remove the need to hard-code the viewname in the Java class).

<bean name="indexController"

         class="org.springframework.web.servlet.mvc.ParameterizableViewController"

          p:viewName="index"  />

**4.**The **UrlFilenameViewController** inspects the URL and retrieves the filename of the file request and uses that as a viewname. For example, the filename of http://www.springframework.org/index.html request is index. (how use it?)

**5. MultiActionController**Spring offers a MultiActionController class that supports the aggregation of multiple request-handling methods into one controller, which then allows you to group related functionality together.

There are two usage-styles for the MultiActionController.

       subclass the MultiActionController and specify the methods that will be resolved by the **MethodNameResolver** on your subclass

       delegate object, on which methods resolved by the **MethodNameResolver** will be invoked.

Method signature which supports multiAction controller.

public (ModelAndView | Map | String | void) actionName(HttpServletRequest request, HttpServletResponse response);

1) if the method return type is Map it looks for a view which is same url name.

Any type of MethodNameResolver  /vtr.htm?method=testmap will looks for vtr.jsp

But the testmap will executed successfully.

public Map testmap(HttpServletRequest arg0, HttpServletResponse arg1) throws Exception {

        return new HashMap();

    }

2) if the method name void response directly written to response Object.

  public void testvoid(HttpServletRequest arg0, HttpServletResponse res,User user) throws Exception {

     res.setContentType("text/xml");

      PrintWriter out= res.getWriter();

      out.write("<ONE>This is multiAction void return type</ONE>");

    }

3) if the method name String then ?????

 public String teststr(HttpServletRequest arg0, HttpServletResponse res) throws Exception {

     return "index";

    }

4) normal signature similar to other controllers third parameter is optional to any of the above method also

public ModelAndView testmap(HttpServletRequest arg0, HttpServletResponse arg1, [command| ExceptionClass] ) throws Exception {

        return new ModelAndView(“index”);

    }

**MethodNameResolver** is responsible for resolving method names based on the specifics of the incoming HttpServletRequest.

|  |  |
| --- | --- |
| **MethodNameResolver** | **Explanation** |
| org.springframework.web.servlet.mvc.multiaction.InternalPathMethodNameResolver | interprets the final filename from the request path and uses that as the method name/ For example, 'http://www.sf.net/testing.view' will result in the methodtesting(HttpServletRequest, HttpServletResponse) being invoked. |
| org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver | interprets a request parameter as the name of the method that is to be invoked.For example, 'http://www.sf.net/index.view?method=testIt' will result in the method testIt(HttpServletRequest, HttpServletResponse) being invoked. The 'paramName' property specifies the name of the request parameter that is to be used. Example to use paramName property shown below example |
| org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver | uses a user-defined Properties object with request URLs mapped to  method names. For example, when the Properties contain '/index/welcome.html=doIt' and a request to  /index/welcome.html comes in, thedoIt(HttpServletRequest, HttpServletResponse) method will be  invoked. |

       Example to use **ParameterMethodNameResolver**

<bean id="paramMultiController"

class="org.springframework.web.servlet.mvc.multiaction.MultiActionController">

<property name="methodNameResolver">

<bean class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">

<property name="paramName" value="method"/>

</bean>

</property>

<property name="delegate">

<bean class="samples.SampleDelegate"/>

</property>

</bean>

       Example to use  **PropertiesMethodNameResolver**

 <bean id="multiController" class="com.myapp.spring.controller.MyMultiController">

 <property name="methodNameResolver">

       <bean class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">

             <property name="mappings">

                   <value>

                       /abc.htm=testmap

                       /mv.htm=testmv

                   </value>

             </property>

       </bean>

  </property>

  </bean>

       Using **MultiActionController** as a Delegate Object method.

<bean id="multiController" class="org.springframework.web.servlet.mvc.multiaction.MultiActionController">

 <property name="methodNameResolver">

      <bean class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">

         <property name="paramName" value="vtr"/>

       </bean>

  </property>

  <property name="delegate">

     <bean class="com.myapp.spring.controller.DelegateObject"/>

  </property>

 </bean>

       **CommandControllers**

|  |  |
| --- | --- |
| **Command Controller** | **Explanation** |
| AbstractCommandController | a command controller you can use to create your own command controller, capable of binding request parameters to a data object you specify. This class does not offer form functionality; it does however offer validation features and lets you specify in the controller itself what to do with the command object that has been populated with request parameter values. |
| AbstractFormController | an abstract controller offering form submission support. Using this controller you can model forms and populate them using a command object you retrieve in the controller. After a user has filled the form, the AbstractFormController binds the fields, validates the command object, and hands the object back to the controller to take the appropriate action. Supported features are: invalid form submission (resubmission), validation, and normal form workflow. You implement methods to determine which views are used for form presentation and success. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the application context |
| SimpleFormController | a form controller that provides even more support when creating a form with a corresponding command object. TheSimpleFormController let's you specify a command object, a  viewname for the form, a viewname for page you want to show the user when form submission has succeeded, and more. |
| AbstractWizardFormController | as the class name suggests, this is an abstract class - your wizard  controller should extend it. This means you have to implement thevalidatePage(), processFinish() and processCancel() methods. |
| BaseCommandController | This class having two subclasses  1) AbstractCommandController  2) AbstractFormController |

**6. BaseCommandController**Controller implementation which creates an object (the command object) on receipt of a request and attempts to populate this object with request parameters.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.  
Properties supported by the BaseCommandController

|  |  |  |
| --- | --- | --- |
| **Name** | **Default** | **description** |
| commandName | command | the name to use when binding the instantiated command class to the request |
| commandClass | null | the class to use upon receiving a request and which to fill using the request parameters. What object is used and whether or not it should be created is defined by extending classes and their configuration properties and methods. |
| validators | null | Array of Validator beans. The validator will be called at appropriate places in the workflow of subclasses (have a look at those for more info) to validate the command object. |
| validator | null | Short-form property for setting only one Validator bean (usually passed in using a <ref bean="beanId"/> property. |
| validateOnBinding | true | Indicates whether or not to validate the command object after the object has been populated with request parameters. |

**7. AbstractCommandController**is required to set command class assetCommandClass() method or set commandClass property in xml file.

<bean class="com.myapp.spring.controller.CommandTwoController">

    <property name="commandClass" value="com.User"/>

  </bean>

                                          (**or)**

public class CommandTwoController extends AbstractCommandController{

    CommandTwoController(){

     setCommandClass(User.class);

    }

    @Override

protected ModelAndView **handle**(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, BindException arg3) throws Exception {

       return new ModelAndView("index","name",((User)arg2).getName());

    }

}

**8. AbstractFormController** : Form controller that 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.

public class FormOneController extends AbstractFormController {

    @Override

  protected ModelAndView **showForm**(HttpServletRequest arg0, HttpServletResponse arg1, BindException arg2) throws Exception {

        return new ModelAndView("index", "name", "showForm");

    }

    @Override

protected ModelAndView **processFormSubmission**(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, BindException arg3) throws Exception {

        System.out.println( " This is  "+  getCommandName());

        return new ModelAndView("index", "name", "processFormSubmission"+arg2 +": " +   ((User)arg2).getName());

    }

}

**Steps involved in the processing this request**

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 (see the properties the superclass exposes), but can also be overridden to e.g. retrieve an object from the database (that needs to be modified using the form).

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. This will render appropriate Strings for those property values, e.g. locale-specific date strings.

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. *Note:* any defined Validators are not applied at this point, to allow partial binding. However be aware that any Binder customizations applied via initBinder() (such asDataBinder.setRequiredFields(String[]) will still apply. As such, if using bindOnNewForm=true and initBinder() customizations are used to validate fields instead of using Validators, in the case that only some fields will be populated for the new form, there will potentially be some bind errors for missing fields in the errors object. Any view (JSP, etc.) that displays binder errors needs to be intelligent and for this case take into account whether it is displaying the initial form view or subsequent post results, skipping error display for the former.

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 (e.g. a List of Locale objects you're going to let the user select one from).

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 theisFormSubmission 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 (e.g. to manually bind request parameters to bean properties, to be seen by the Validator).

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 object.

13.   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).

14.   Call processFormSubmission() to process the submission, with or without binding errors. This method has to be implemented in subclasses.

In session form mode, a submission without an existing form object in the session is considered invalid, like in case of a resubmit/reload by the browser. The [handleInvalidSubmit](http://www.blogger.com/Frameworks/api/org/springframework/web/servlet/mvc/AbstractFormController.html" \l "handleInvalidSubmit%28javax.servlet.http.HttpServletRequest,%20javax.servlet.http.HttpServletResponse%29) method is invoked then, by default trying to resubmit. It can be overridden in subclasses to show corresponding messages or to redirect to a new form, in order to avoid duplicate submissions. The form object in the session can be considered a transaction token in that case.

**Configuration Properties**

|  |  |  |
| --- | --- | --- |
| **Name** | **Default** | **Description** |
| bindOnNewForm | false | Indicates whether to bind servlet request parameters when creating a new form. Otherwise, the parameters will only be bound on form submission attempts. |
| sessionForm | false | Indicates whether the form object should be kept in the session when a user asks for a new form. This allows you e.g. to retrieve an object from the database, let the user edit it, and then persist it again. Otherwise, a new command object will be created for each request (even when showing the form again after validation errors). |

**9. SimpleFormController**It is similar to  AbstractFormController. But there are some main difference I found are described here

       we no need to  implement showForm and processFormSubmission:

       A form view and a success view can be configured declaratively.

       It supports three difference **onSubmit** overloaded methods. The preference order they called shown below.

                                I.      protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object command, BindException errors) throws Exception;

                             II.      protected ModelAndView onSubmit(Object command, BindException errors) throws Exception;

                           III.      protected ModelAndView onSubmit(Object command) throws Exception;

**Steps Involved in the processing requests**

**I.**Call to processFormSubmission which inspects the Errors object to see if any errors have occurred during binding and validation.

           II.      If errors occured, the controller will return the configured formView, showing the form again (possibly rendering according error messages).

         III.      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 invokeonFormChange(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.

          IV.      If no errors occurred, the controller will call onSubmit using all parameters, which in case of the default implementation delegates toonSubmit 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.

**Properties that needs to be configured**

|  |  |  |
| --- | --- | --- |
| **Name** | **Default** | **Description** |
| formView | null | Indicates what view to use when the user asks for a new form or when validation errors have occurred on form submission. |
| successView | null | Indicates what view to use when successful form submissions have occurred. Such a success view could e.g. display a submission summary. More sophisticated actions can be implemented by overriding one of the onSubmit() methods. |

**10.** **AbstractWizardFormController** - as the class name suggests, this is an abstract class - your wizard controller should extend it. This means you have to implement the validatePage(), processFinish() and processCancel() methods.

<bean class="com.myapp.spring.controller.WizardController">

   <property name="commandClass" value="com.User"/>

   <property name="pages">

       <list>

            <value>redirect:formtwo.htm</value>

           <value>redirect:formone.htm</value>

           <value>redirect:formone.htm</value>

       </list>

   </property>

 </bean>

**11. ThrowawayController(interface) :** it is specifically intended for controllers that are not aware of the Servlet API at all.

public class ThrowAwayController implements  ThrowawayController {

    public ModelAndView execute() throws Exception {

        System.out.println("execute");

        return new ModelAndView("index");

    }

}

**HANDLER MAPPINGS**

       When a request comes in, the DispatcherServlet will hand it over to the handler mapping to let it inspect the request and come up with an appropriate HandlerExecutionChain. Then the DispatcherServlet will execute the handler and interceptors in the chain (if any).

       The concept of configurable handler mappings that can optionally contain interceptors (executed before or after the actual handler was executed, or both) is extremely powerful. A lot of supporting functionality can be built into customHandlerMappings. Think of a custom handler mapping that chooses a handler not only based on the **URL** of the request coming in, but also on a specific state of the session associated with the request.

There are two Spring’s most commonly used handler mappings.They both extend the*org.springframework.web.servlet.handler.AbstractHandlerMapping* and share the following properties.

|  |  |
| --- | --- |
| **Property** | **Description** |
| interceptors | the list of interceptors to use. |
| defaultHandler | the default handler to use, when this handler mapping does not result in a matching handler. |
| order( see example 5) | based on the value of the order property (see theorg.springframework.core.Ordered interface), Spring will sort all handler mappings available in the context and apply the first matching handler. |
| alwaysUseFullPath | if this property is set to true, Spring will use the full path within the current servlet context to find an appropriate handler. If this property is set to false (the default), the path within the current servlet mapping will be used. For example, if a servlet is mapped using /testing/\* and the alwaysUseFullPath property is set to true,/testing/viewPage.html would be used, whereas if the property is set to false,/viewPage.html would be used. |
| urlDecode | the default value for this property is true, as of Spring 2.5. If you prefer to compare encoded paths, switch this flag to false. However, note that theHttpServletRequest always exposes the servlet path in decoded form. Be aware that the servlet path will not match when compared with encoded paths. |
| lazyInitHandlers | allows for lazy initialization of *singleton*handlers (prototype handlers are always lazily initialized). Default value is false |

**Handler Mappings**

|  |  |
| --- | --- |
| **Handler Mapping** | **Description** |
| org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping | It maps Controller to URLs that are based on the Controller’s bean name |
| org.springframework.web.servlet.handler.SimpleUrlHandlerMapping | Maps Controllers to URLs using a property collection defined in the spring application context. |
| org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping | It maps Controller to URLs by using the controller’s class name as the basis for the URL. |
| org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping | Map controllers to URLs using source-level metadata placed in the controller code.The metadata is defined using Jakarta commons attributes |
| org.springframework.web.servlet.handler.AbstractUrlHandlerMapping |  |

**Examples to use Handler Mappings**

1. **Using SimpleUrlHandlerMapping**

<bean id=”simpleUrlMapping” class=”org.sf.web.servlet.handler.SimpleUrlHandlerMapping”>

<property name=”mappings”>

    <props>

<prop key=”/home.htm”>homeController</prop>

<prop key=”/index.htm”>indexController</prop>

    </props>

</property>

</bean>

1. **Using BeanNameUrlHandlerMapping**

<bean id=”urlMapping” class=”org.sf.web.servlet.handler. BeanNameUrlHandlerMapping”/>

<bean name=”/index.htm” class=”com.IndexController”/>

1. **Using ControllerClassNameHandlerMapping**

<bean id=”urlMapping” class=” org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping”/>

<bean class=”com.IndexController”/>

1. **Using CommonsPathMapHandlerMapping**

<bean id=”urlMapping” class=”org org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping”/>

/\*\*

\*@@org.springframework.web.servlet.handler.commonsattributes.PathMap(“/home.htm”)

\*/

Public class HomePageController extends AbstractController{

//  implement **handleRequestInternal**method

}

1. **Working with multiple HandlerMapping**

<bean id=”beanNameUrlMapping” class=”org.sf.web.servlet.handler. BeanNameUrlHandlerMapping”>

<property name=”order”><value>**1**</value></property>

</bean>

<bean id=”simpleUrlMapping” class=”org.sf.web.servlet.handler.SimpleUrlHandlerMapping”>

<property name=”order”><value>**0**</value></property>

<property name=”mappings”>

    <props>

<prop key=”/home.htm”>homeController</prop>

<prop key=”/index.htm”>indexController</prop>

    </props>

</property>

</bean>