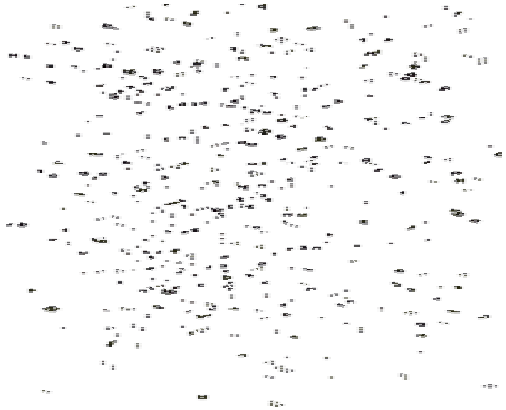
Copyright Notice

Welcome...



Spring Framework 5.3.1

Prakash Badhe

Prakash.badhe@vishwasoft.in

About Me

- ✓Technology Trainer, Mentor for 20+ years.
- ✓Worked as Director Technology Practices - AgileSoft Methodologies
- ✓Promoting Agile technical practices and processes.
- ✓Web Technologies consultant and mentor
- ✓Passion for technologies and frameworks
- ✓Worked on medium and large sized dynamic web projects.
- ✓Specialized in java technologies and frameworks
- ✓Proficient with Html5,Ajax,XML standards
- ✓Working with JavaScript libraries and frameworks. .

About you..

- **Prerequisite** : Conversant in Java and Java EE web applications and programming.
- Job Role and skill-Sets
- Training Objectives
- Your job experience in Spring Framework
- Awareness about Java EE programming
- Prior experience with Web MVC frameworks

Agenda

- Spring Framework 5.1.3 Features
- Spring Containers and services
- Spring IoC/Dependency Injection
- Spring Aspect Oriented Programming
- Spring Data Access overview
- Spring ORM usage
- Spring MVC Web Applications
- Spring Web Templates
- Spring JavaEE support features.
- Build REST Services with Spring Framework
- Grails Web Applications Overview

Set up

- Windows 10/Ubuntu/Mac 64 bit
- JDK 1.8
- Chrome/firefox/safari latest browsers
- Spring Tool Suite/Eclipse-Jee-Mars
- Apache Tomcat 8.5 Web Server
- Spring Framework 5.3.1and dependencies zip
- Hibernate 4.2
- MySQL DB Server 8.0
- MySQL Workbench client 8.0
- Adobe pdf reader
- Open internet connection

What is Library ?

What is Framework..?

What is Spring ?

- Spring is an open source java framework.
- Spring is developed by ***Interface21***.
- Spring comes in two flavors
  - Java based framework –Spring5.x
  - .NET based framework –Spring. Net

Goal of Spring

- Reduce the complexity of J2EE.
- Concentrate on the OO design than the implementation technology.
- Not compete with good existing solutions, but should foster integration.
- Provide service abstraction layers.
- Support modular approach and reduce code *plumbing*.

Nature of Spring

- Spring is a *non-invasive* i.e. unlike struts where your classes are tightly coupled with specific API and hence limits the portability to another framework.
- Spring deals with POJO Based components.
- Spring is *portable* framework that allows you to introduce as much or as little as you want to your application.
- Promotes decoupling and reusability
- Allows developers to focus more on reused business logic and less on plumbing problems.

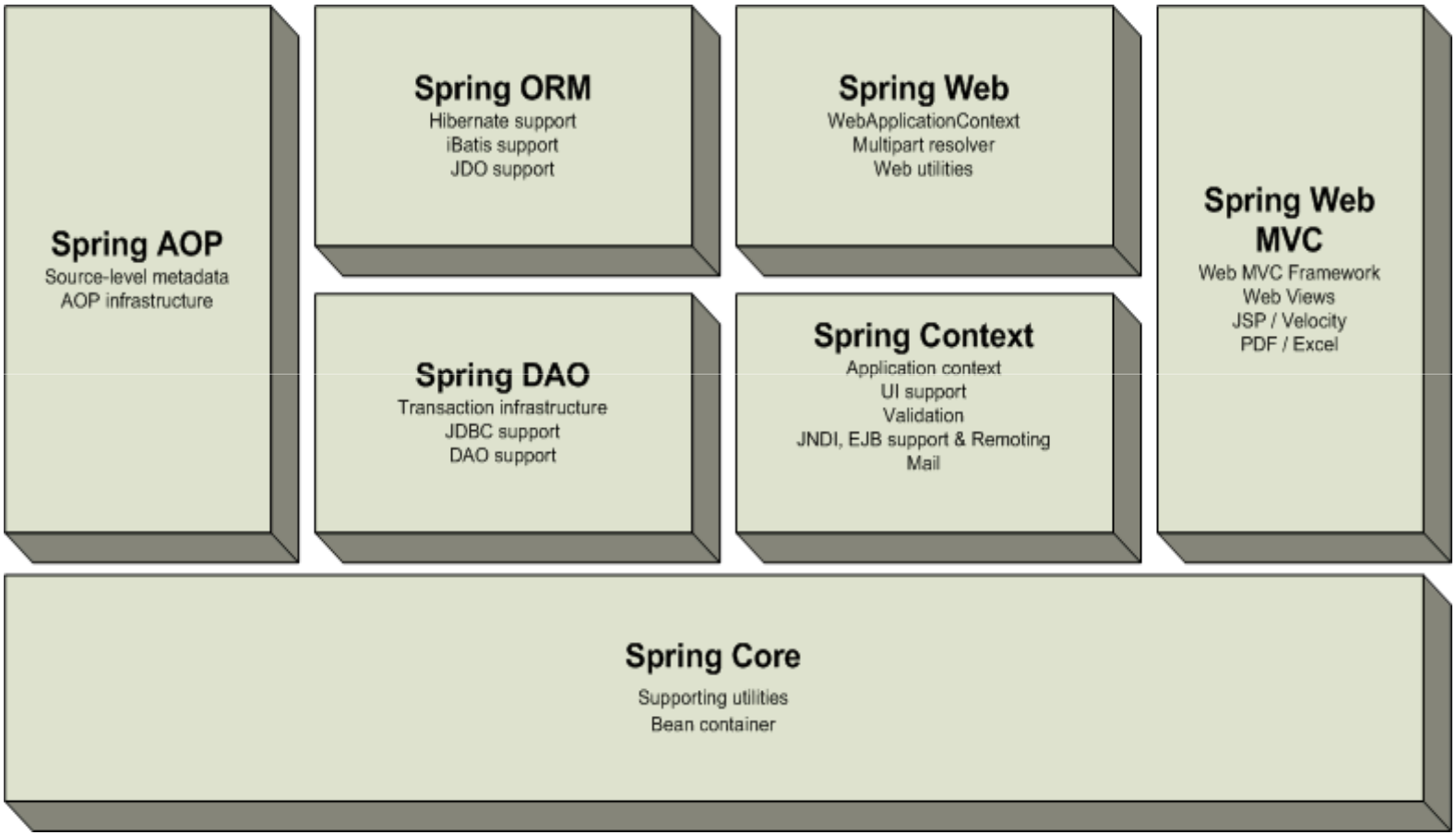
Spring Framework Features

- Inversion of Control (IoC) container or Dependency Injection
- AOP Support.
- Web MVC support.
- Integration with other web frameworks.
- Simplify Database access.
- Integration with ORM frameworks like Hibernate,iBatis ,TopLink.
- Testing support
- Simplify programming with EJB,RMI,JMS,Web Services.
- Integration with any type of java application project i.e desktop. Mobile, web application or enterprise application projects

What is new in Spring 5 ?

- Fully supports Java 8 features.
- Removed Deprecated Packages and Methods
- Support for Java EE 6 and 7.
- Groovy Bean Definitions and DSL support
- New REST controller
- Asynchronous REST Client requests
- Web socket support
- Server side events sent support
- Different ORM layers support
- Testing improvements

Spring Architecture



How much to use from Spring ?

- Spring is not an *all-or-nothing* solution
  - Can be a one stop shop or just use certain subs
  - The Spring distribution comes as one big jar file and alternatively as a series of smaller jars broken out for different modules (so you can include only what you need)
- Spring can also be considered a library
  - Implementation classes are considered as a public API
  - Twenty isolated Spring modules for different functionalities.

Spring Features

- The Java 5,6,7 and Java 8 are fully supported
- The modules are organized in separate jar files.
- Source annotations in java files
- Spring Expression Language
- IoC enhancements
- General-purpose type conversion
- Object to XML mapping support
- Support for REST web services
- @MVC additions in annotations
- Declarative model validation
- Java EE 6 early support
- Embedded database support

What's new in Spring ?

- The version 4.0 is the latest major release of the Spring Framework and the first to fully support Java 8 features.
- The minimum required JRE is Java SE 6
- Java EE version 6 or above is now the baseline for Spring Framework 4, with the JPA 2.1 and Servlet 3.0 specifications being of particular relevance.
- Hibernate 4.3 is a JPA 2.1 provider and supported as of Spring Framework 4.0.
- Supports the Groovy DSL for groovy applications.
- The @RestController with Spring MVC applications.
- The AsyncRestTemplate class for non-blocking asynchronous support in REST clients.

Spring 5.3 new features

- The support for WebSocket-based, two-way communication between client and server in web applications.
- Supports higher level message-oriented programming model on top of WebSocket that's based on SockJS and includes STOMP sub-protocol support.
- New features for use in unit and integration testing.
- JMS listener annotations
- Synchronous request-reply support in JmsTemplate.
- Supports JCache (JSR-107) annotations.
- The HttpMessageConverter options for json, Gson and Google Protocol Buffers.

Spring 5.3.1 Features

- Multiple bean Profiles can be added to bean configuration classes and xml definitions for the same application module.
- The JSP links to controllers by referring to controller mappings by name.
- New messaging module with many types carried over from the Spring Integration.
- project.
- Support for JSR-310: Data and Time API.
- Generalized support for conditional bean creation.
- Support for many JEE specs including JMS 2.0, JTA 1.2, JPA 2.1 and Bean Validation 1.1.

Spring IoC/DI

What is dependency ?

Dependency example

- Jsp with java bean.
- <jsp:useBean is=“bean” class=“com.app.MyBean”>
- <jsp:setProperty name = "bean" property = “uname" value = “Nana"/> <jsp:getProperty name = "bean" property = “uname"/>

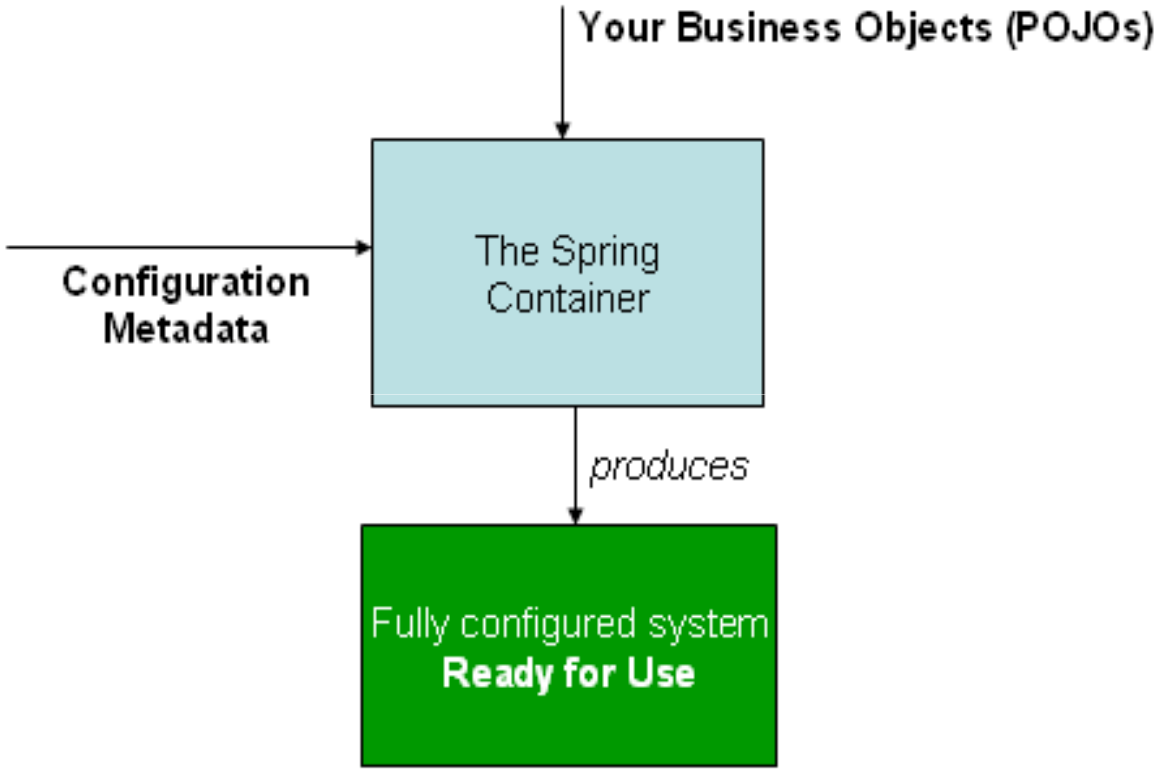
Dependency Injection/IoC

- Inversion Of Control.. IoC is in short '*the object sharing and life cycle management'* service provided by the Spring container.
- You can compare this to the Java Beans object creation and scope management offered by the java web server (container) for Java Server pages in web applications.
- The IoC container provides all the facilities required for creating a POJO object i.e. initializing the POJO properties which may be simple types, collections or other java objects which may be POJO again, i.e. the IoC *injects the dependencies* into the objects and hence the term **Dependency Injection**.

How Dependency Injection works ?

- The IoC container offers these services to POJO objects which are not tied to any specific framework.
- The POJO object configuration can be programmatic or declarative.
- These POJO objects managed by the container are called as beans.

Container Injection control



Inject dependencies in IoC Way…

- **Property Injection** : Create the POJO/Java Bean objects by using default constructors and injecting the dependent properties by calling the *setter methods* for properties injection.
- **Constructor Injection** : Create the bean objects by using parameterized constructors and injecting the dependent properties though the *constructor*.
- **Method Injection** :Implement the method behaviour dynamically.

Spring IoC

- The ***BeanFactory*** interface standardizes the Spring bean container behavior.
- It provides the basic client view of a bean container which is responsible for creating and managing the life cycle of POJO bean objects.
- The number of implementations such as **XmlBeanFactory** , **XmlWebApplicationContext** , **FileXmlApplicationContext** are responsible for managing the POJO objects in different contexts.

The IoC/DI Container

- In declarative configuration the POJO objects are defined with unique identities (*id* and/or *name*) are defined by specifying the class types and certain property values are specified in xml format. These are called as bean definitions.
- These property values specified can be simple types,collections,certain java objects or other POJO beans.
- The DI container at runtime identifies the POJO bean definitions inside the *beans* root of xml config file or with source annotations and create, manage the bean objects and return them to the applications.

Sample Bean.xml

```
<beans>
   <!-- injecting built-in vals sample -->
   <bean id="Simple" class="User">
      <property name="name">
         <value>John Smith</value>
      </property>
      <property name="age">
         <value>35</value>
      </property>
      <property name="height">
         <value>1.78</value>
      </property>
   </bean>

   <!-- oracle bean implemented by Bookworm class and  used by other beans -->
   <bean id="ora" class="Bookworm"/>

   <!-- inject ref using name -->
   <bean id="MyRef" class="TestRef">
      <property name="dataSys">
         <ref local="ora"/>
      </property>
   </bean-->
</beans>
```

How to access the DI Container ?

```
//Create the FileResource
FileResource  fir = new FileResource ("bean.xml");
//Create the beanFactory object
XmlBeanFactory  xb = new XmlBeanFactory(fir);
//Request the 'User' object from the beanFactory
User u=(User)xb.getBean("MyBean");
//Use the User 'u' object.
//The object 'u' will be destroyed by the beanFactory
    when its scope ends.
```

Bean Definitions Examples

- The bean class is the actual implementation of the bean being described by the BeanFactory.
- Bean examples – DAO, DataSource, Transaction Manager, Persistence Managers, Service objects, etc
- Spring config contains implementation classes while your code should program to interfaces.
- Bean behaviors supported by Bean (IoC) Container:
  - Singleton or prototype
  - Autowiring
  - Initialization and destruction methods
    - init-method
    - destroy-method
- Beans can be configured to have property values set.
  - Can read simple values, collections, maps, references to other beans, etc.

Factory to create bean objects

- User defined factory classes with static methods which return either current or other objects can be configured in xml configuration file.
- The same can be achieved by having a instance method in a factory class. This is useful for creating objects dynamically at runtime.

Bean Object Life cycle

- While creating the bean objects by the DI Container the bean life cycle methods can be specified via
  - Standard life cycle interface
  - Declarative configuration in XML/Source annotations

Bean Object Scope

- Define the sharing scope of bean objects i.e. Singleton or Prototype (for each request new bean object is served to application). By default it is Singleton behavior unless specified.
- The bean life cycle can be controlled by defining certain call back methods (for initialization and destroying) of Spring interfaces in POJO classes. This becomes **invasive** programming).

Wiring the beans…

- The process of resolving the dependencies on other beans by looking in the xml configuration and injecting dependencies is called wiring.
- This can be done automatically, thereby reduce or eliminate the need to specify properties or constructor arguments in xml configuration.

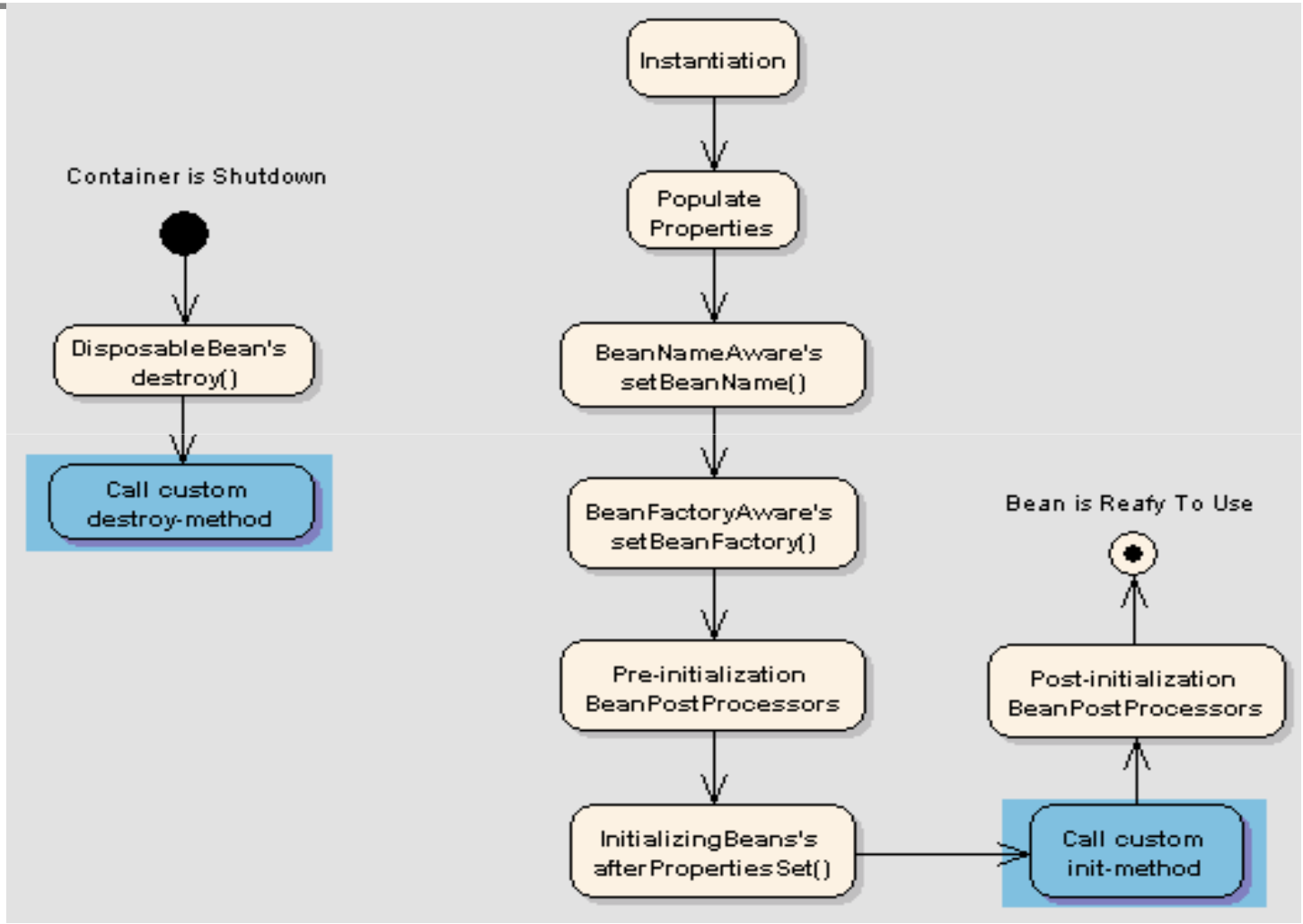
Auto wiring the beans

- In XML-based configuration metadata, the autowire mode for a bean definition is specified by using the autowire attribute of the <bean/> element. The following values are allowed:
- *No* : No autowiring at all. Bean references must be defined via a ref element. This is the default.
- *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.

More on Auto wiring..

- *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.
- *Constructor* : 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.
- *Default*: The default mode can be configured in global application context for all beans in the same context.

Bean Life Cycle



Bean Life cycle support

- Spring bean factory is responsible for managing the life cycle of beans created through spring container.
- The life cycle of beans consist of **call back methods** which can be categorized broadly in two groups:
  - Post initialization call back methods
  - Pre destruction call back methods

Bean Life Control

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

1. InitializingBean and DisposableBean callback interfaces
2. Other Aware interfaces for specific behavior.
3. custom init() and destroy() methods in bean configuration file
4. @PostConstruct and @PreDestroy annotations

Spring FactoryBean classes

- Factory beans are mostly used to implement framework facilities.
  - When looking up an object (such as a data source) from JNDI, you can use JndiObjectFactoryBean.
  - When using classic Spring AOP to create a proxy for a bean, you can use ProxyFactoryBean.
  - When creating a Hibernate session factory in the IoC container, you can use LocalSessionFactoryBean.
- In most cases, you rarely have to write any custom factory beans.

Spring bean scope values

- The core of spring framework is it's bean factory and mechanisms to create and manage such beans inside Spring container.
- The beans in spring container can be created in five scopes. .
  - singleton
  - prototype
  - request
  - session
  - global-session: in Portlet applications.

Spring bean post processor

- A bean post processor allows additional processing before and after the bean initialization callback method.
- The main characteristic of a bean post processor is that it will process all the bean instances in the IoC container one by one, not just a single bean instance.
- The bean post processors are used for checking the validity of bean properties or altering bean properties according to certain criteria.
- To create a bean post processor class in spring, implement the BeanPostProcessor interface and register in the context configuration.

DI Annotations

- @Configuration : Load the values from properties/config files
- @Bean : Declare bean methods on current class
- @DependsOn : Declare dependency on another resource/bean
- @Primary :
- @Lazy : Declare lazy initialization
- @Import : load @Bean definitions from another configuration class
- @ImportResource : import external xml config files
- @Value : Specify property value
- @PostConstruct : Bean life cycle method post construct

More Annotations

- @PreDestroy : Bean life cycle method before destroy
- @Required : applies to bean property setter methods
- @Resource : To add external resource
- @Inject : to inject the resource in bean object
- @Autowired : Autowire the bean
- @Component
- @Service

Enable component scanning

- The java source level annotation will be scanned and configured only when they are scanned by DI container of spring framework.
- To enable this scanning, to use ‘context:component-scan’ tag in the applicationContext.xml .
- The context:component-scan element requires a base-package attribute, which, as its name suggests, specifies a starting point for a recursive component search.
- You may not want to give the top package for scanning to spring, so declare multiple component-scan elements, each with a base-package attribute pointing to a different package.

Autowired and Resource

- The @Autowired annotation tries to find a bean of type class specified in the spring context and then inject the same into parent bean.
- Similar to this is @Resource annotation that tries to find the bean with the name specified.
- To summarize the @Autowired wires by type and @Resource wires by name.

Qualifier and Required

- In case of same type of more than one bean found in auto-wiring,the spring context throws BeanInitializationException.
- To resolve the auto-wiring ambiguity, the @Qualifier annotation tries to inject a Class type bean whose qualifier value is specified.
- To ensure that a property is injected with a value, use the @Required annotation. It is equivalent to auto wire check.

Bean Life Cycle annotations

- The method marked with @PreDestroy is called on the bean object just after the bean is destroyed.
- The method marked with @PostConstruct is called before de-initializing the bean.

Stereotyping Annotations

- These annotations are used to stereotype classes with regard to the application tier that they belong to.
- Classes that annotated with one of these annotations will automatically be registered in the Spring application context if <context:component-scan> is in the Spring XML configuration.
- @Component :The Generic stereotype annotation for any Spring-managed component and automatically registers beans in Spring context.The name can be specified for the bean by passing it as the value of @Component.
- @Component(“myBean") public class Accouny{ ... }

More annotations

- @Repository :Stereotypes a component as a repository. Also indicates that SQLExceptions thrown from the component's methods should be translated into DataAccessExceptions, Used in persistence applications.
- @Service : Stereotypes a component as a service to be injected into parent beans.

Property Editors

- While setting the property values in xml config file for a POJO bean class, the values are always specified as character string values.
- What if the property value is an integer, date(mm/dd/yyyyy format) or an another bean object ?
- The built in Property Editors supported by Spring make this value conversion to required type.
- You can also define your own Property Editors to customize the property value conversions.

Dependent beans

- A bean is a dependency of another bean is expressed by the fact that one bean is set as a property of another. This is typically done with the <ref/> element in XML-based configuration metadata of beans.
- To specify indirect dependencies 'depends-on' attribute is used to express a dependency on a single another bean.
- This will trigger the initialization of the dependent bean first and then for depending bean.

Initialize dependent beans…

- The 'depends-on' attribute also specifies init time dependency .
- Dependant beans that are defined in the 'depends-on' attribute will be initialized *first* prior to the relevant bean itself being initialized.
- This allows to control the initialization order for different beans.

Spring AOP

OO Design is good but…

- The *oop* languages makes the code design more reusable and compact.
- But sometimes it makes the code scattered in number of modules.
- More code plumbing has to be done…
- Example : adding message logging in each of class methods makes the logging code scattered in number of methods and classes.
- This makes code maintenance and updation a tedious work.

An AOP Example

- Try-Catch block
  - int num =0;
  - try{
  - int data = 100/num;
  - }
  - catch(Exception e)
  - { }

AOP on server side

- Error handling in JSP
- <%@ page errorPage="errorHandler.jsp" %>
- Servlet and Filter mapping
- servlet-mapping>
  <servlet-name>milkServlet</servlet-name>
  <url-pattern>/drink/*</url-pattern>
  </servlet-mapping>

AOP is the solution…

- 'Aspect Oriented Programming(AOP)' addresses these issues.
- Instead of adding the logging code in each of the methods, you delegate the logging functionality (*concern*) across number of methods to an aop *container* and specify which methods require logging applied. The container will call the logging aspect of the code when the methods are invoked.

AOP

- Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. In addition to classes, AOP gives you *aspects*.
- Aspects enable modularization of concerns (required additional business logic) such as transaction management that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns.)

AOP Examples

- For exception handling in the JSP pages, instead of writing individual 'try-catch' blocks in each jsp we make a common exception handler jsp and specify in each jsp the ref. of this common handler by using *isErrorPage* and *errorPage* attributes in page directive tag.
- When any of the jsp page generates the exception it is forwarded to common handler page by the web container.
- Another example is specifying Filters to be invoked transparently before Servlet invocations in java web applications.

AOP advantages

- Complements OO programming.
- Clean and modular code without additional code plumbing.
- Aspects can be added or removed as needed without changing the code.
- The business object code is not tied to specific API or framework

Some AOP terminologies

- **Concern** - The required functionality.(logging,tracing etc.)
- **Aspect** – A modularization of a concern that cuts across multiple objects and methods.
- **Join point** – A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.
- **Advice** – the block of code that runs based on the pointcut definition
- **Pointcut** – A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).
- **Advisor** - references the advice and the pointcut.
- **Weaving** – Linking aspects with other application types or objects to create an advised object can be done at runtime or compile time. Which inserts the advice (crosscutting concerns) into the code. Spring provides run time weaving.

More aop terms..

- **Target object**: Object being advised by one or more aspects. Also referred to as the advised object. Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.
- **AOP proxy**: An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on).
- **Introduction**: Declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any proxied object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching.

Spring AOP support

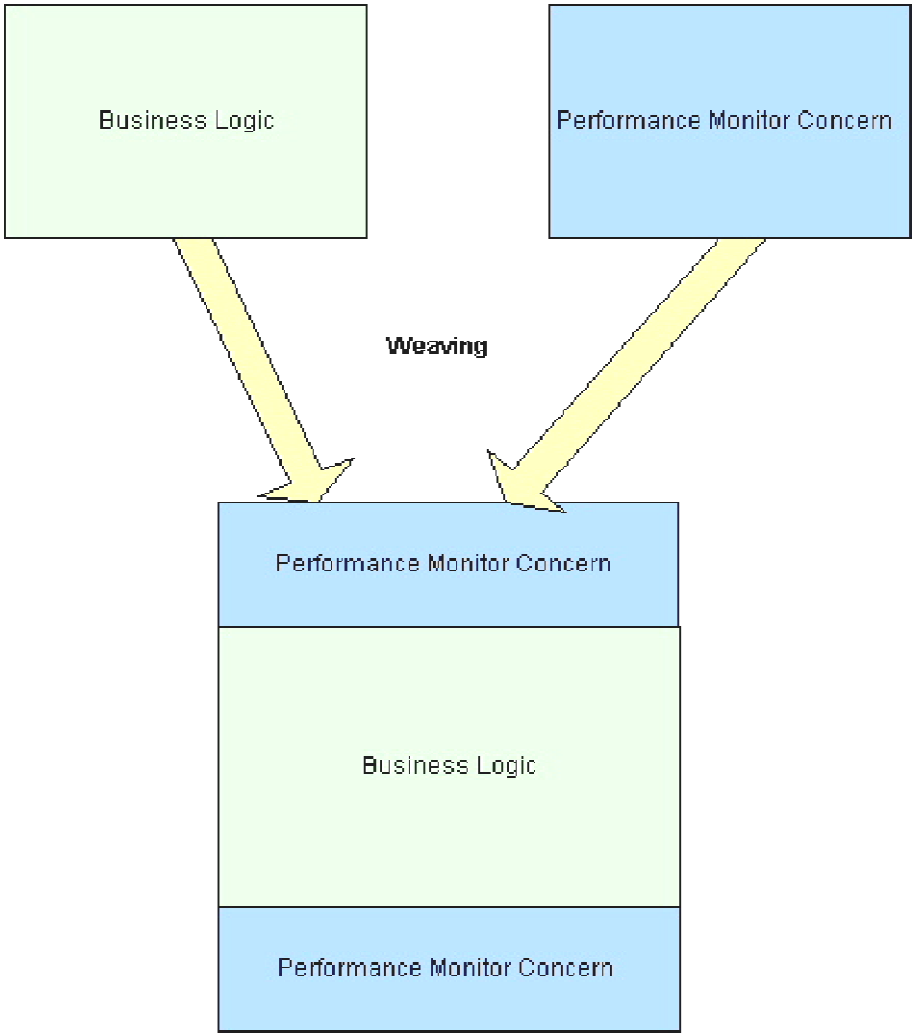
- AOP Framework that builds on the aopalliance interfaces.
- Aspects are coded with pure Java code.
- Spring aspects can be configured using its own IoC container.
  - Objects obtained from the IoC container can be transparently advised based on configuration
- Spring AOP has built in aspects such as providing transaction management, performance monitoring and more for your beans.
- Spring supports proxy based AOP to bean methods.

Spring Advices

Spring AOP supports the following advices:

- *Before advice*: (Advice is applied before method is executed)
- *After returning advice*: (Advice is applied after method is returned normally without throwing any Exceptions)
- *After throwing advice*: Advice to be executed if a method exits by throwing an exception.
- *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
- *Around advice*: Advice that surrounds a join point such as a method invocation. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Simple AOP Weaving



Applying the Advices..

- Aspects are weaved together at runtime. AspectJ uses compile time weaving.
- Spring AOP also includes advisors that contain advice and pointcut filtering.
- IoC + AOP is a great combination that Makes the code *non-invasive*

Using Spring AOP

- To provide declarative enterprise services, especially as a replacement for EJB declarative services such as *declarative transaction management*.
- Providing declarative linking of web application components.
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

Spring AOP Features

- Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a J2EE web container or application server or in a desktop application.
- Spring AOP currently supports only method execution join points.
- The aspects can be programmatically or declaratively configured through xml.
- The aspects can be configured with jdk1.5 annotations features.

Spring Advice interfaces

- BeforeAdvice
  - Public void before(Method m, Object[] args, Object target) throws Throwable;
- ThrowsAdvice
  - public void afterThrowing(Method m, Object[ ] args, Object target, Throwable ex);
- AfterReturningAdvice
  - Public void afterReturning(Object returnValue, Method m, Object[ ] args, Object target) throws Throwable;

Method Interceptor

- MethodInterceptor used to implement for around Advice on methods.
  - public Object invoke(MethodInvocation invocation) throws Throwable

Defining PointCut

```
package org.springframework.aop;
public interface Pointcut
{
 ClassFilter getClassFilter();
 MethodMatcher getMethodMatcher();
}
```

PointCut model

- Used to target advices to particular classes and methods.
- Spring's pointcut model enables pointcut reuse independent of advice types.
- It's possible to target different advices using the same pointcut.
- ClassFilter and MethodMatcher are interfaces which are used to restrict the pointcut to a given set of target classes and used to test matching of a particular method against a given method, respectively.

Operations on pointcuts

- Spring supports two operations on pointcuts:
  - Union means the methods that either pointcut matches and is usually more useful.
  - Intersection means the methods that both pointcuts match.
- Pointcuts can be composed using the static methods in the *org.springframework.aop.support.Pointcuts* class, or using the *ComposablePointcut* class in the same package.

Static pointcuts

- Static pointcuts are based on method and target class, and cannot take into account the method's arguments.
- Static pointcuts are sufficient for most usage. It's possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Springs Advisors

- Spring has a *RegexpMethodPointcutAdvisor*, that allows us to also reference an Advice (Advice can be an interceptor, before advice, throws advice etc.)
- This advice is mapped to specific matching methods.

Advice declarations

```
<bean id="tg"
  class="org..aop.support.RegexpMethodPointcutAdvisor">
<property name="advice">
   <ref local="beanAdvice"/>
</property>
  <property name="patterns">
      <list>
      <value>.*set.*</value>
      <value>.*Calculate</value>
      </list>
  </property>
</bean>
```

Dynamic pointcuts

- These take into account method *arguments*, as well as static information. This means that they must be evaluated with *every* method invocation and the result cannot be cached, as arguments will vary.
- Example is the control flow pointcut.
- Control flow pointcuts are specified using the ControlFlowPointcut class.
- Spring provides useful pointcut super classes to help you to implement your own pointcuts.

Cross cutting concerns

- Concerns applied across number of classes are called as cross cutting.

Spring AOP Annotations

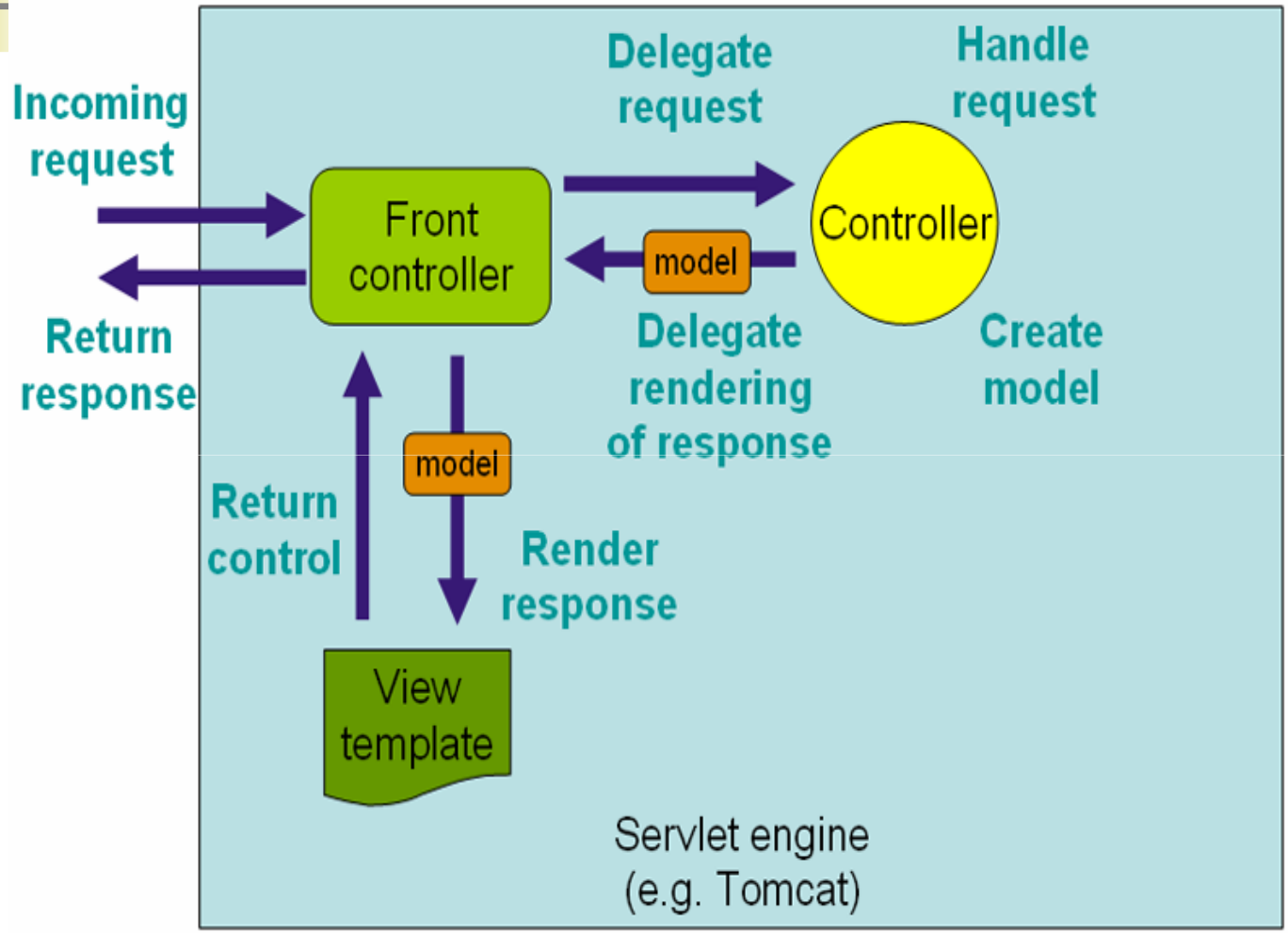
@Pointcut : Define a pointcut class
@Aspect : Define an aspect bean class
@After : After advice method
@AfterReturning : Method to be called after method returns normally
@Before : Before method advice
@Around : Around method advice
@Target : Target object
@AfterThrowing : Exception advice

Spring Web MVC

Spring MVC Features

- Spring's own MVC structure providing more modularity.
- Spring’s own custom tag library to be used in jsp.
- This MVC used in web applications as well as standalone graphic applications.
- Configuration not tied to specific presentation technology.
- Supports localization, view resolvers, custom tag library etc.
- Supports client side validations.
- IoC and AOP support can be wired in mvc configurations.

Spring MVC framework



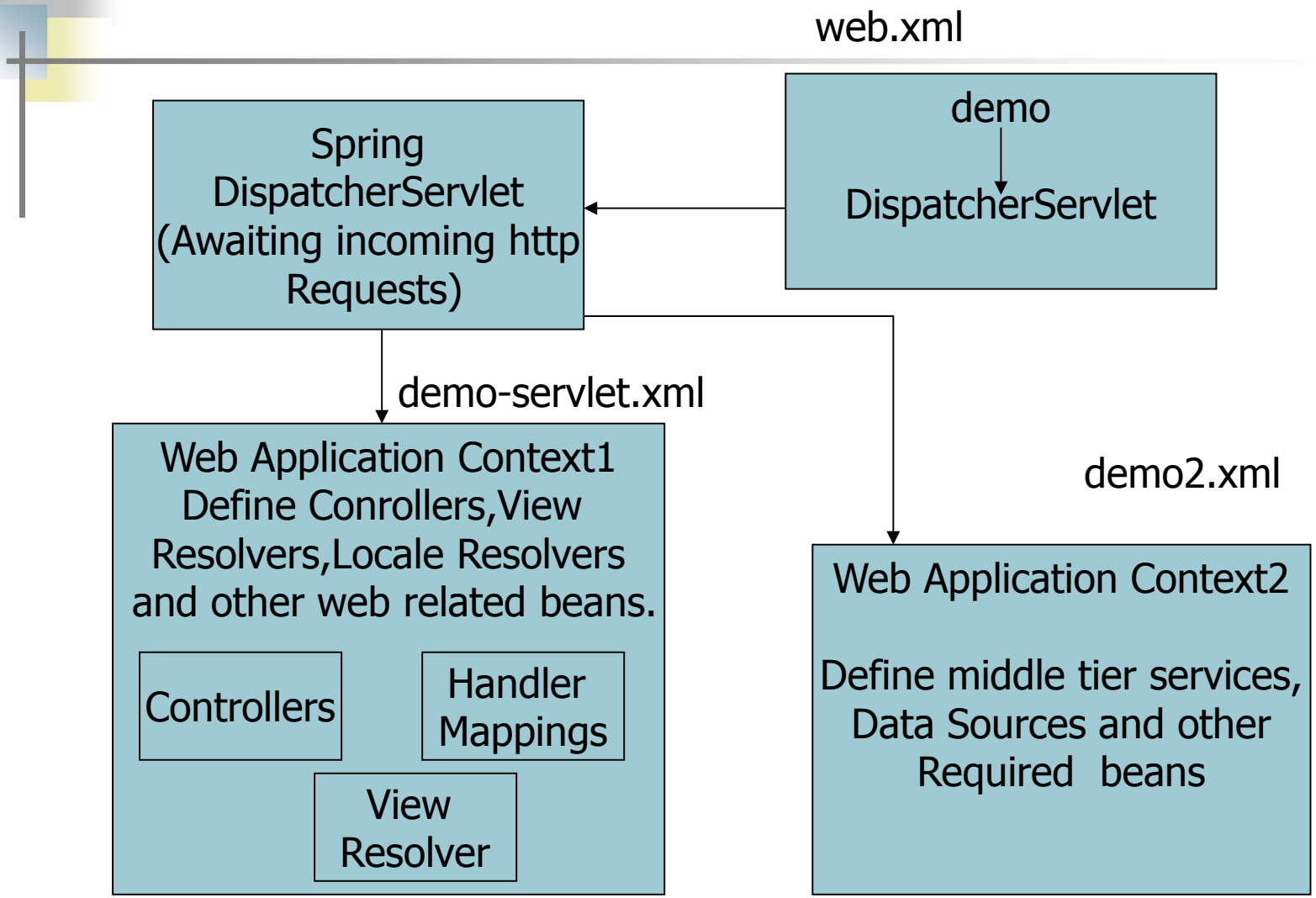
Spring MVC Components

- Spring Separates the Dispatcher and Controller behavior in two different components.
- DispatcherServlet responsible for intercepting the request and dispatching for specific urls. Analogous to ActionServlet of Struts framework.
- Controller interface implementations specifies user defined model, view coordination, equivalent to *Action* in Struts.
- ModelAndView class objects encapsulates view and model linking.
- Any Java Object can be used as Command object (FormBean in Struts).

Mapping for DispatcherServlet

- Requests that are to be handled by the DispatcherServlet are to be mapped using a URL mapping in the web.xml config file.
- Each DispatcherServlet has its own *WebApplicationContext* and the specific *bean* objects used in this context and the *Controllers* are to be defined in context(name mapped to in web.xml) specific servlet.xml file.

Dispatcher Contexts



Dispatcher mapping in web.xml

```
<web-app>
   <servlet>
      <servlet-name>demo</servlet-name>
   <servlet-class>
       org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
  </servlet>
   <servlet-mapping>
    <servlet-name>demo</servlet-name>
    <url-pattern>*.form</url-pattern>
   </servlet-mapping>
</web-app>
```

demo-servlet.xml

```
<beans>
  <!-- Controller class mappings →
  <bean id="DemoControl" class="MyController"/>
  <bean id="HelloControl" class="HelloController">
    <property name="productManager">
      <ref bean="prodMan"/>
    </property>
  </bean>
<bean id="urlMapping" class=
"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/work.htm">DemoControl</prop>
        <prop key="/hello">HelloControl</prop>
      </props>
    </property>
</bean> <beans>
```

beans for the Dispatcher

- **Controllers**: classes that implement Controller.
- **Handler mappings**: handle the execution of a list of pre- and post-processors and controllers that will be executed if they match certain criteria.
- **View Resolvers**: resolves view names to views. (JSPs)
- **Locale Resolver** : resolves the locale a client is using and loading the specific property files (through ResourceBundle) for internationalized views.
- **Theme resolver** : Class capable of resolving themes your web application can use, for example, to offer personalized layouts.
- **Handler exception resolver**: offer functionality to map exceptions to views orimplement other more complex exception handling code.
- **multipart file resolver :** offers the functionality to process file uploads from HTML forms.

The ‘C’ in MVC

- Controllers provide access to the application behavior which is typically defined by a service interface.
- Controllers interpret user input and transform this input into a specific model which will be represented to the user by the view.
- Spring provides a base interface *Controller* and wide variety of its implementations.

Controller interface

```
package org.springframework.web.servlet.mvc;
public interface Controller
{
 public ModelAndView handleRequest(HttpServletRequest
    request, HttpServletResponse response) throws
    Exception;

/* process the request and return a ModelAndView object
    which the DispatcherServlet will render.*/

}
```

Spring's Controllers

- *AbstractController* : abstract base class.
- *ParameterizableViewController*: returns view name that is defined in the web application context (No hard coding of view name).
- *UrlFilenameViewController*: inspects the URL and retrieves the filename of the file request and uses that as a view name. e.g. the filename of http://server.com/index.html request is index.
- *MultiActionController*: groups multiple actions into one controller.

Command controllers

- Command controllers: provide a way to interact with data objects and dynamically bind parameters from the HttpServletRequest to the data object specified.
- These data objects are similar to Struts ActionForm, but in Spring, they don't have to extend from a framework-specific class.

More Controllers

- *AbstractCommandController*: used to create your own command controller. Capable of binding request parameters to a data object specified.
- It offers validation features and lets you specify in the controller what to do with the command object that has been populated with request parameter values.

Form Controllers

- Spring provides couple of Form Controllers to deal with client side data validation and submission to action url.
- AbstractFormController : base class to define your own form controller and add validation features.
- SimpleFormController: supports creating a form with a corresponding command object used as FormBean.

Wizard view in web ?

- The *AbstractWizardFormController* supports wizard like view in web page.
- Extend this class and override couple of methods to provide custom wizard view.

Model : the 'M' in MVC

- The model is generally defined as a Map that can contain objects that are to be displayed in the View.
- The **ModelAndView** object encapsulates the relationship between view and model and is returned by corresponding Controller methods.
- The ModelAndView class uses a **ModelMap** class that is a custom Map implementation where valuse are added in key-value fashion.

Actions on Model

- The Controller generates the model object and forwards it to view page.
- Before forwarding validation and command object population will be done by Controller.
- The model map values and command object properties can be accessed in view pages and presented to the user.

View : the presentation

- The view page can be explicitly returned as part of *ModelAndView* object by the controller
- In case of mapping logical name of view can be resolved to particular view page in case the ModelAndView doesn't contain the view reference.(i.e. null value)
- The view name can be independent of view technology (i.e. without using naming '.jsp' in controller) and *resolved* to specific technology by using *View Resolver and rendered by View*.

View Page mapping

- The controller can return a logical view name, which a *view resolver* resolves to a particular view name and technology and uses view class to render it.
- So the application doesn't tie you to a specific view technology. (i.e. jsp,velocity etc.- no hard coded references to view pages instead it will just return the file name of the view minus its file extension).
- This makes faster portability to other view technologies.

View Resolvers in Spring

- Spring's view resolvers
- The most commonly used view resolvers are
  - InternalResourceViewResolver
  - ResourceBundleViewResolver
- A custom view resolver can be defined by implementing ViewResolver and View interfaces.

View Classes

- The view class is responsible for rendering the view in the browser.
- Spring provides view classes for JSPs, Velocity templates and XSLT views.

InternalResourceViewResolver

```
<bean id="viewResolver2“ class=
   "org..web..view.InternalResourceViewResolver">
<property name="viewClass" value=
   "org.springframework.web.servlet.view.JstlView"/>
<property name="prefix" value="/WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
</bean>
```

ResourceBundleViewResolver

```
<bean id="viewResolver"
    class="org..web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"/>
 </bean>
```

```
# And a properties file is uses (views. properties in WEB-
    INF/classes):
#These properties file will be specific to client locale.

welcome.class=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp
productList.class=org.springframework.web.servlet.view.JstlView
productList.url=/WEB-INF/jsp/productlist.jsp
```

The different types of views can be mixed by using only one resolver.

Chaining View Resolvers

- Spring supports more than just one view resolver. This allows you to chain resolvers and, for example, override specific views in certain circumstances.
- Chaining view resolvers is straightforward - just add more than one resolver to your application context and, if necessary, set the order property to specify an order.
- The higher the order property, the later the view resolver will be positioned in the chain.

Internationalization in Spring

- *DispatcherServlet* offers to automatically resolve messages using the client's locale and is done with *LocaleResolver* objects.
- The *LocaleResolver* can also change the current request Locale.
- Another way : *ResourceBundleMessageSource* bean can be configured with ***messageSource*** id and having the property file base name configured.
- The Locale resolvers are configured in application context of DispatcherServlet .
- An interceptor in the handler mapping can be attached to the request to change the locale under specific circumstances, based on request parameter.

Default View name resolving

- The URL http://localhost/registration.html will result in logical view name of *'registration'* being generated by the *DefaultRequestToViewNameTranslator*.
- This logical view name will then be resolved into the '/WEB-INF/jsp/registration.jsp' view by the *InternalResourceViewResolver* bean.
- Advantage is hide the actual view names from the web user.

Default mapping

- Spring has the default initialization of *DefaultRequestToViewNameTranslator*, even if not configured explicitly in context.
- The DispatcherServlet will actually instantiate an instance of this class if one is not explicitly configured.
- Mapping for InternalResourceViewResolver must be provided which will specify the type and location of view pages.

Handler mappings

- Using a handler mapping incoming web requests are mapped to appropriate handlers. These mappings can contain filter-interceptors or url mapping handlers.
- A HandlerExecutionChain will be constructed by the DispatcherServlet which contains the handler that matches the incoming request, and may also contain a list of handler interceptors that are applied to the request.
- The DispatcherServlet will execute the handler and interceptors in the chain (executed before or after the actual handler was executed, or both).

Bean Name mapping

```
<beans>
<bean id="handlerMapping" class=
   "org..web.servlet.handler.BeanNameUrlHandlerMapping
   " />
<bean name="/UpdateAcount.form"
   class="org.springframework.web.servlet.mvc.SimpleFormCo
   ntroller">
     <property name="formView" value="account"/>
     <property name="successView" value="account-created"/>
     <property name="commandName" value="account"/>
     <property name="commandClass" value="bank.Account"/>
</bean>
<beans>
```

Default Handler Mapping

- If no handler mapping is found in the context, the DispatcherServlet creates a default *BeanNameUrlHandlerMapping*.

Mapping from Anywhere ?

- The ***SimpleUrlHandlerMapping*** handler routes requests for any '/*/url-defined' in any directory to the specified 'Controller', which can be a SimpleFormController or UrlFilenameViewController which will direct the request to specific named view page.

Request Interceptors

- For applying specific functionality to certain requests, *HandlerInterceptor* can be implemented.
- This interface defines three methods, one that will be called before the actual handler will be executed, one that will be called after the handler is executed, and one that is called after the complete request has finished.

MVC Annotations

- ✓@Controller
- ✓@RequestMapping
- ✓@SessionAttributes
- ✓@RequestParam
- ✓@ModelAttribute
- ✓@ResponseBody
- ✓@CookieValue
- ✓@ExceptionHandler
- ✓@InitBinder
- ✓ @Valid

Custom Handler

- 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 can be defined.
- AbstractHandlerMapping supports custom handler mappings by extending it.
- BeanNameUrlHandlerMapping : maps incoming HTTP requests to names of beans, defined in the web application context.

Themes

- A theme is a collection of static resources affecting the visual style of the application, typically style sheets and images.
- Spring enhances the user web experience by allowing the look and feel of the application to be themed.
- The themes are created by ResourceBundleThemeSource object(default) which ,loads the locale specific property files.

ThemeSource

- You register a bean in the application context with the reserved name "themeSource", with the class name as ResourceBundleThemeSource and base prefix property as the name of property file.
- The web application context will automatically detect that bean and start using it by loading the specific theme source property.
- ThemeSource property contents
  - styleSheet=/themes/style.css
  - background=/themes/img/Back.jpg

Loading the themes

- The spring custom tag library provides spring:theme custom tag to load the specific theme contents.
- DispatcherServlet looks for a bean named "themeResolver" to find out which ThemeResolver implementation to use. A theme resolver detects the theme that should be used for a particular request and can also alter the request's theme.

Applying the theme

```
<%@ taglib prefix="spring"
   uri="http://www.springframework.org/tags"%>
<html>
  <head>
    <link rel="stylesheet" href="<spring:theme
    code="styleSheet"/>"  type="text/css"/>
  </head>
 <body background="<spring:theme
    code="background"/>"
  ...
 </body>
</html>
```

ThemeResolver implementations

- **FixedThemeResolver** : Selects a fixed theme, set using the "defaultThemeName" property.
- **SessionThemeResolver**: The theme is maintained in the users HTTP session. It only needs to be set once for each session, but is not persisted between sessions.
- **CookieThemeResolver** :The selected theme is stored in a cookie on the user's machine.
- Spring also provides a **ThemeChangeInterceptor**, which allows changing the theme on every request by including a simple request parameter.

Spring's form tag library

- Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC.
- Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and Generate the html code.
- This form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with.
- These form tags make JSPs easier to develop, read and maintain.

Handling exceptions

- Spring provides HandlerExceptionResolvers to ease the pain of unexpected exceptions occurring while the request is being handled by a controller which matched the request.
- These provide information about what handler was executing when the exception was thrown.
- Furthermore, a programmatic way of handling exception gives many more options for how to respond appropriately before the request is forwarded to another URL.
- The HandlerExceptionResolver interface specifies the behavior for this hnadler.

Default Configuration

- In the web application context you can reduce the mapping code by using spring default configurations options supported.
- For example using the Spring bean *ControllerClassNameHandlerMapping* to map the controllers., will map the *HomeController* to the '/home*' request URL and so on.

Integration with View Technologies

Spring Tiles support

- It is possible to integrate Struts-Tiles - just as any other view technology - in web applications using Spring.
- The tiles definitions will be loaded by Spring bean class 'TilesConfigurer'.
- The tiles definitions can be reolved and rendered by view bean classes 'TilesView' or 'TilesJstlView' for jsp with tiles and jstl tags.

Tiles Loading

```
<bean id="tilesLoader"
   class="org.springframework.web.servlet.view.
   tiles.TilesConfigurer">

<property name="definitions">
     <list>
    <value>/WEB-INF/defs/ TileDefs.xml</value>
    <value>/WEB-INF/defs/ templates.xml</value>
    </list>
</property>
</bean>
```

Tiles View

```
<bean id="viewResolver"
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="viewClass"
  value="org.springframework.web.servlet.view.tiles.TilesView"/>
</bean>
```

XSL Transformation in view

- XSLT is a transformation language for XML and is popular as a view technology within web applications.
- XSLT can be a good choice as a view technology if your application deals with XML data.
- Spring supports XML model data transformed with XSLT as view in a Spring Web MVC application.

XSLT Transform in Spring

- Spring class AbstractXsltView allows to extend and override createXsltSource(..) method to generate xml source data.
- To specify additional xml data parameters, override the getParameters() method of the AbstractXsltView class and return a Map of the name/value pairs.
- Define view.properties file in /WEB-INF/classes directory.
- Define stylesheet in /WEB-INF/xsl directory so that it is not exposed to the web user.

XSL View properties

- home.class=xslt.MyXSLContoller
- home.stylesheetLocation=/WEB-INF/xsl/style.xslt
- home.root=words
- #‘words’ is generated xml root element
- # xslt.MyXSLContoller is controller extending *AbstractXsltView* class.

PDF and excel Views

- Spring also supports to generate a PDF document or an Excel spreadsheet dynamically from the model data.
- The document is the view and will be streamed from the server with the correct content type to enable the client PC to run their spreadsheet or PDF viewer application in response.
- In order to use Excel views, you need to add the 'poi' library to your classpath, and for PDF generation, the ‘iText.jar’. Both are included in the main Spring distribution.

Data Access with Spring

Spring Data Access Features

- Spring offers higher level data access classes with couple of methods for easier data access and manipulation.
- Supports various call back interfaces to define your own data manipulation.
- Classes to represent RDBMS queries, updates, and stored procedures as thread safe, reusable objects.
- Utility classes for easy *DataSource* access and various simple DataSource implementations that can be used for testing and running unmodified JDBC code outside of a J2EE container.

JdbcTemplate for data access

- Simplifies the use of JDBC since it handles the creation and release of resources. This helps to avoid common errors such as always *forgetting* to close the connection.
- It executes the core JDBC workflow like statement creation and execution, leaving application code only to provide SQL and extract results.
- It executes SQL queries, update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values.
- It catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy.
- A JdbcTemplate instance is threadsafe.

SQL error codes for different DBs?

- Spring contains SQLException translation functionality for any kind of sql error generated from any supported database s.
- Spring also offers translation of database-specific SQL error codes to meaningful exception classes.
- The code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling.
- All translated exceptions are unchecked giving the option of catching the exceptions that you can recover from while allowing other exceptions to be propagated to the caller.

Jdbc Callbacks…

- 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, providing SQL and any necessary parameters.
- The *CallableStatementCreator* interface creates callable statement.
- The *RowCallbackHandler* interface extracts values from each row of a ResultSet
- The RowMapper interface implementation maps a particular record with certain object and returns it.
- The Spring IoC and AOP features can be applied.

DataSource

- A DataSource is part of the JDBC specification and can be seen as a generalized connection factory.
- It allows a container or a framework to hide connection pooling and transaction management issues from the application code.
- Spring's JDBC layer provides a data source from JNDI or from the user defined configuration.
- *DriverManagerDataSource* is the Spring implementation for DataSource.

DataSource from JNDI

- Spring provides 'JndiObjectFactoryBean' for fetching a DataSource from JNDI and give the DataSource bean reference to other beans.
- Switching to another DataSource/Database becomes just a matter of configuration.

Injecting Dependencies for JdbcTemplate

```
<beans>
<bean id="corporateDao" class="com.CorporateDao">
   <property name="dataSource" ref="dataBean"/>
</bean>
<!-- the DataSource for configuration  -->
<bean id="dataBean" destroy-method="close"
    class="org.apache.commons.dbcp.BasicDataSource">
      <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>
</beans>
```

SimpleJdbcTemplate

- The SimpleJdbcTemplate class is a wrapper around the classic JdbcTemplate that takes advantage of Java 5 language features such as varargs and autoboxing.
- Offers a much smaller subset of the methods exposed on the JdbcTemplate class.

SingleConnectionDataSource

- The SingleConnectionDataSource class is an implementation of the SmartDataSource interface which extends DataSource.
- *Provides a single* Connection that is *not* closed after use.
- Obviously, this is not multi-threading capable.

TransactionAwareDataSourceProxy

- Is a proxy for a target DataSource, which wraps that target DataSource to add awareness of Spring-managed transactions.
- It is similar to a transactional JNDI DataSource as provided by a J2EE server.

JDBC operations as Java objects

- Spring contains classes that allow to access the database in a more object-oriented manner.
- For example, one can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object.
- One can also execute stored procedures and run update, delete and insert statements with java objects!

Java Object-rdbms classes

- **SqlQuery**
- **MappingSqlQuery**
- **SqlUpdate**
- **StoredProcedure**
- **SqlFunction**

Transaction annotations

- ✓@Transactional
- ✓@TransactionConfiguration
- ✓@Rollback
- ✓@AfterTransaction
- ✓@BeforeTransaction
- ✓@NotTransactional

Spring ORM Support

ORM Support

- Spring Framework provides integration with the ORM frameworks.
  - *Hibernate,*
  - *JDO*
  - *Oracle TopLink*
  - *iBATIS SQL Maps*
  - *JPA*
- Spring offers them resource management, DAO implementation support, and transaction strategies.

ORM Integration strategies

- Using Spring's DAO 'templates‘ classes.
- Coding DAOs against underlying ORM APIs.
- In both cases, the DAOs can be configured through Dependency Injection and participate in Spring's resource and transaction management.

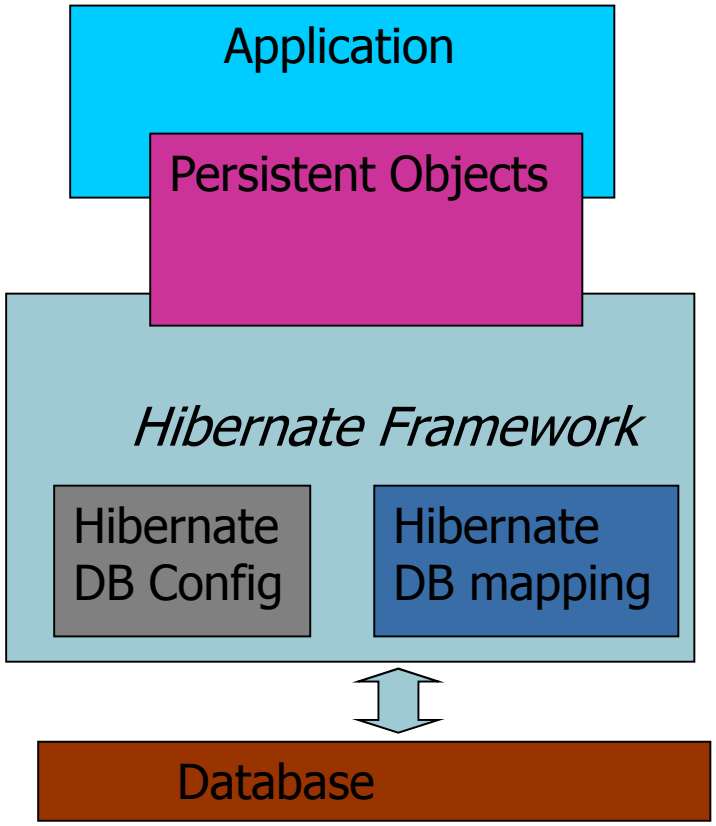
Spring DB approach

- All the individual data access features are usable on their own but integrate nicely with Spring's application context concept, providing XML-based configuration and cross-referencing of plain JavaBean instances that don't need to be Spring-aware.
- In a typical Spring app, many important objects are JavaBeans: data access templates, data access objects (that use the templates), transaction managers, business services are independent of Spring.

Spring advantages

- Offers clear application layering with any data access and transaction technology.
- Offers loose coupling of application objects.
- No more business service dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. And so on.

Hibernate ORM Framework



Hibernate Framework Components

- *SessionFactory*: A thread safe object consisting of jdbc mappings with a single database configured with xml.
- *Session*: A single-threaded, short-lived object representing a communication between the application and the persistent store.
- *Persistent objects and collections*: Short-lived, single threaded objects containing persistent state and business functions. It is mapped to DB structure using xml
- *Transaction*: A single-threaded, short-lived object used by the application to specify atomic units of work. Abstracts application from underlying JDBC transaction. A Session might span several Transactions in some cases.

Spring Hibernate Support

- Spring provides ***HibernateTemplate*** and *HibernateDaoSupport* classes to work with Hibernate ORM framework.
- The *SessionFactory* configuration can be programmatic or declarative in Spring beans xml format which offers declarative IoC and AOP support also.

HibernateTemplate in Spring

- This class provides many methods that mirror the methods exposed on the Hibernate Session interface, in addition to a number of convenience methods.
- HibernateTemplate will ensure that Session instances are properly opened and closed and automatically participate in transactions.
- The template instances are thread-safe and reusable.

HibernateTemplate methods

- For actions like a find, load, saveOrUpdate, or delete operations, HibernateTemplate offers alternative convenience methods.
- To customize the data manipulation or queries which are not possible directly with HibernateTemplate methods, **HibernateCallback** interface specifies methods which can be implemented by custom classes.

Hibernate Config with Spring

```
<beans>
    <bean id="mySource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
            <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
            <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
            <property name="username" value="sa"/>
            <property name="password" value=""/>
    </bean>
    <bean id="mySessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
            <property name="dataSource" ref="mySource"/>
            <property name="mappingResources">
                <list>
                <value>product.hbm.xml</value>
                </list>
            </property>
              <property name="hibernateProperties">
                <value>
                  hibernate.dialect=org.hibernate.dialect.HSQLDialect
                </value>
            </property>
    </bean>
</beans>
```

DataSource Switching…

- Switching from a local Jakarta Commons DBCP BasicDataSource to a JNDI-located DataSource (managed by an application server) is just a matter of configuration:
- <beans>

  <bean id="myDataSource" class="org.springframework.jndi.***JndiObjectFactoryBean***">

  <property name="jndiName"

  value="java:comp/env/jdbc/myds"/>

  </bean>

  </beans>
- A JNDI-located SessionFactory can also be accessed using Spring's JndiObjectFactoryBean.

SQLExceptionTranslator

- Specifies the translation between SQLExceptions (specific to database) and Spring's own data-access DataAccessException.
- The Implementations can be generic (using SQLState codes for JDBC) or DataBase specific (for example, using Oracle error codes) for greater precision.
- Implemented by *SQLErrorCodeSQLExceptionTranslator*.
- Use with JdbcTemplate to translate SQL errors.

Spring DAO Support

- The HibernateDaoSupport class offers methods to access the current transactional Session and to convert exceptions.
- User defined DAO class can extend ***HibernateDaoSupport*** and override required methods without callbacks and work with HibernateTemplate object.

Spring Transaction Support

- Spring Framework supports the comprehensive transaction support via a consistent abstraction for transaction management.
- Spring transactions can be applied to any class in any environment and in any context i.e. not just the EJB or the J2ee container or global or local in the same way.
- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate,JPA, and JDO.
- Supports declarative and programmatic transaction management.

More transaction features…

- Spring provides simpler API for programmatic transaction management.
- Spring Transaction can be configured with jdk 1.5 style annotations in class code itself.
- Rollback rules can be supplied declaratively.
- Transaction support for supported ORM frameworks,JMS and application server specific like WebLogic , Websphere Servers.

Transaction abstraction

- *PlatformTransactionManager* is central interface in Spring's transaction infrastructure specifying rollback, commit and other methods.
- DataSourceTransactionManager and JtaTransactionManager are some of the implementations.
- *TransactionStatus* interface represents the status of a transaction.
- *TransactionDefinition* interface specifies Isolation, transaction propagation, Timeout and ReadOnly features.

DataSourceTransactionManager

- A PlatformTransactionManager implementation for single JDBC datasources. It binds a JDBC connection from the specified data source to the currently executing thread, allowing for one thread connection per data source.
- Supports custom isolation levels, and timeouts that get applied as appropriate JDBC statement query timeouts.
- This can be used instead of JtaTransactionManager in the single resource case, as it does not require the container to support JTA.

DataSourceTransactionManager

```
<bean id="data" 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.DataSourceTransactionMan
    ager">
   <property name="dataSource" ref="data"/>
</bean>
```

Using JtaTransactionManager

- <jee:jndi-lookup id="dataSource" jndi-name="jdbc/store"/>
- <bean id="txManager" class= "org.springframework.transaction.jta.JtaTransactionManager" />
- Here a container DataSource, obtained via JNDI, is used in conjunction with Spring's JtaTransactionManager.
- If the DataSource is managed by the J2EE container it should be *non-transactional* as the Spring Framework will manage transactions.

HibernateTransactionManager

```
<bean id="MySessionFactory"
    class="org..orm.hibernate.LocalSessionFactoryBean">
     <property name="dataSource" ref="dataSource" />
     <property name="mappingResources">
          <list>
          <value>myData.hbm.xml</value>
          </list>
     </property>…. </property>
</bean>
<bean id="txManager"
    class="org.springframework.orm.hibernate.HibernateTransactionMan
    ager">
 <property name="sessionFactory" ref=" MySessionFactory " />
</bean>
```

- For Hibernate and JTA transactions combined, the JtaTransactionManager can be used as with JDBC or any other resource.

Spring advantage…

- In all these transaction cases, application code will not need to change at all whenever we change the database or the environment.
- We can change how transactions are managed merely by changing configuration.

Transaction management strategy

- Programmatic transaction management is usually a good idea only if you have a small number of transactional operations.
- if the application has numerous transactional operations, declarative transaction management is the best choice.

Testing Support in Spring

Testing Support in Spring

- The Spring team considers developer testing to be an absolutely integral part of enterprise software development.
- The Dependency Injection provides one more benefit that your code should really depend far *less* on the container than in traditional J2EE development.
- This facilitates easier unit testing on POJO bean objects.
- The POJOs that comprise your application should be testable in JUnit tests, with objects simply instantiated using the new operator, *without Spring or any other container*.

Easier unit testing

- Spring's clean layering and componentization of your codebase will naturally facilitate easier unit testing. That makes easier test driven development (TDD)
- For example, you will be able to test Data service layer objects by stubbing or mocking DAO interfaces, without any need to access the persistent data while running unit tests.
- Spring also supports testing annotations (java 1.5) to enable testing without writing specialized testcases.

Integration Testing

- The Spring will provide you to test the correct wiring of your Spring IoC container beans without requiring to deploy in the j2ee container.
- In case of data access using JDBC or an ORM tool- you can test the correctness of SQL statements / or Hibernate XML mapping .
- The Spring Framework provides support for integration testing in the form of the ***mock*** classes that are packaged in the spring-mock.jar library.
- Spring also provides couple of valuable JUnit TestCase super classes for integration testing.

Spring Test Case features

- Spring IoC container caching between test case execution.
- The transparent Dependency Injection of test fixture instances.
- Transaction management appropriate to integration testing.
- A number of Spring-specific inherited instance variables that are really useful during integration testing.

Transaction management test

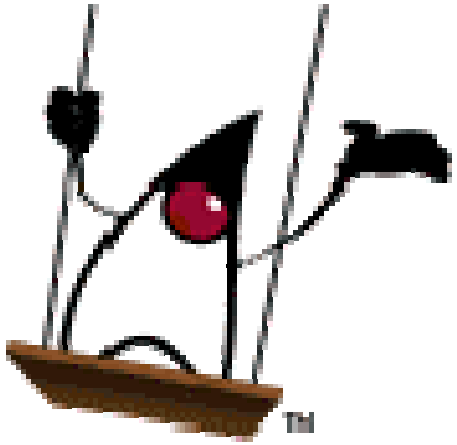
- The Spring classes in org.springframework.test package create and roll back a transaction for each test.
- You simply write code that can assume the existence of a transaction.
- To override rollback or commit the transactions you can write code to test the Transaction management by overriding particular methods in these test classes.

Other Spring features

- Spring Scheduler
- Support for Remoting and web services.
- Support for writing EJB.
- JMS Messaging with Spring.
- Portlet development support
- Spring IDE a plug-in of Spring for fast development with eclipse IDE.

Spring Resources

- www.springframework.org
- Spring Forums.
- www.Sourforge.net



Thank You!