The Spring Framework: An Open Source Java

Platform for Developing Robust Java Applications

**Dashrath Mane, Ketaki Chitnis, Namrata Ojha**

***摘要*— *本文介绍了Spring框架的基本概念。Spring框架是一个提供了全面基础架构支持的开源的Java平台，可以非常轻松、快速地开发健壮的Java应用程序。Spring框架是一个轻量级的解决方案，是构建企业级应用的潜在的一站式服务。***

***索引术语— 面向切面编程, 依赖注入, Ioc 容器, ORM.***

# 一、引言

Spring是企业级Java最流行的应用程序开发框架。全球数百万开发者使用Spring框架创建高性能、易测试、可复用的代码。Spring框架是一个开源的Java平台，最初由Rod Johnson编写，并于2003年6月首次根据Apache2.0许可发布。

Spring在大小和透明度方面是轻量级的。最初版本的Spring大约为2MB。Spring框架的核心功能可用于开发任何Java应用程序，但也有用作在Java EE平台上构建Web应用程序的扩展。Spring框架的目标是通过启用基于POJO的编程模型，使J2EE开发变得简易，促进良好的编程实践。Spring框架为现代基于Java的企业级应用程序提供了全面的编程和配置模型-适用于任何类型的部署平台。Spring的一个关键要素是应用程序级别的基础设施支持：Spring专注于企业级应用程序的“管道”，以便团队可以专注于应用程序级别上的业务逻辑，无需与特定部署环境产生不必要的联系。Spring包括：

* 使用基于XML注释的配置文件进行灵活的依赖注入。
* 使用基于代理和基于AspectJ的变体，对面向切面编程提供高级支持。
* 对Hibernate和Quartz等常见开源框架提供一流的支持。
* 构建RESTful MVC应用程序和服务端的灵活的Web框架。

**Manuscript published on 30 July 2013.**

**\***Correspondence Author(s)

**Mr Dashrath Mane**, Assistant Professor, Department of MCA, V.E.S. Institute of Technology, Mumbai, India.

**Miss NamrataOjha**, Final Year MCA Student, V.E.S. Institute of Technology, Mumbai, India.

**Miss KetakiChitnis**, Final Year MCA Student, V.E.S. Institute of Technology, Mumbai, India.

© The Authors. Published by Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP). This is an [open access](https://www.openaccess.nl/en/open-publications) article under the

CC-BY-NC-ND licens[e http://creativecommons.org/licenses/by-nc-nd/4.0/](http://creativecommons.org/licenses/by-nc-nd/4.0/)

Spring采用模块化设计，允许逐步采用各个部分，例如核心容器或JDBC支持。虽然所有的Spring服务都非常适合Spring核心容器，但也有许多服务可以在容器之外以编程方式使用。

支持的部署平台包括独立应用程序、Tomcat和WebSphere等Java EE服务器。Spring也是主要云平台上支持的Java的一等公民，例如Heroku、Google App Engine、Amazon Elastic Beanstalk和VMware的Cloud Foundry。[1]

# 二、Spring框架架构

Spring可能成为您所有企业应用程序的一站服务；然而Spring是模块化的，允许您提供适合您的模块，而不必引入其余模块。

Spring框架提供了大约20个模块，可以根据应用程序需求使用。

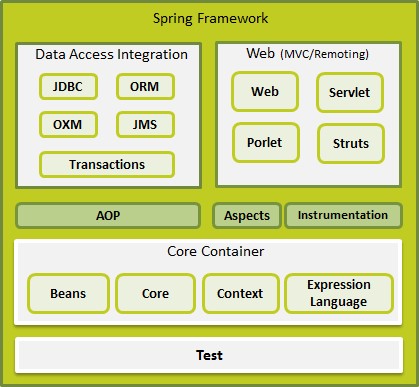


图1. Spring框架结构

## A. 核心容器

核心容器由Core、Beans、Context和Expression Language模块组成，其详细信息如下：

* Core模块提供了框架的基本部分，包括Ioc（控制反转）和依赖注入功能。
* Beans模块 提供了BeanFactory，它是工厂模式的一个复杂实现。
* Context模块建立在核心和Beans模块提供的坚实基础之上，它是访问任何已定义和配置对象的媒介。ApplicationContext接口是Context模块的焦点。
* Expression Language模块提供了一种强大的表达式语言，用于在运行时查询和操作对象图。

## B. 数据访问/集成

数据访问/集成层由JDBC、ORM、OXM、JMS和Transaction模块组成，其详细信息如下：

* JDBC模块提供了一个JDBC抽象层，从而无需进行繁琐的JDBC相关编码。
* ORM模块为流行的对象关系映射API提供了集成层，包括JPA、JDO、Hibernate和iBatis。
* OXM模块提供了一个抽象层，支持JAXB、Castor、XMLBeans、JiBX和XStream的对象/XML映射实现。
* JMS服务（Java Message Service，Java消息服务）。JMS模块包含生成和使用消息的功能。
* Transcation模块支持针对实现特殊接口的类和所有POJO的编程式和声明式事务管理。

## C. 网络

Web层由Web、Web-Servlet、Web-Struts、Web-Portlet模块组成，具体模块如下：

* Web模块提供了基本的面向Web的集成功能，例如多部分文件上传功能，以及通过Servlet监听器 和面向Web的应用程序上下文来初始化Ioc容器。
* Web-Servlet模块包含Spring针对Web应用程序的模型视图控制器（MVC）实现。
* Web-Struts模块包含在Spring应用程序中集成经典Struts Web层的支持类。

***D. 其他***

* AOP模块提供面向切面的编程实现，允许您定义方法拦截器和切入点来干净地解耦应分离功能的代码。
* Aspects模块提供与AspectJ的集成，AspectJ是一个强大且成熟的面向切面编程（AOP）框架。
* Instrumentation模块提供类检测支持和类加载器实现，供特定应用程序使用。
* Test模块支持使用JUnit或TestNG框架测试Spring组件。

# 三、Spring IOC容器

Spring容器是Spring框架的核心。容器将创建对象、将它们连接在一起、配置它们并管理从创建到销毁的完整生命周期。Spring容器使用依赖注入（DI）来管理组成应用程序的组件。这些对象称为Spring Bean，我们将在下一章中讨论。

容器通过读取提供的配置元数据来获取有关实例化、配置和组装哪些对象的指令。配置元数据可以用XML、Java注释或Java代码表示。下图是Spring工作原理的高级视图。Spring Ioc容器利用Java POJO类和配置元数据生成完全配置且和执行的系统或应用程序。

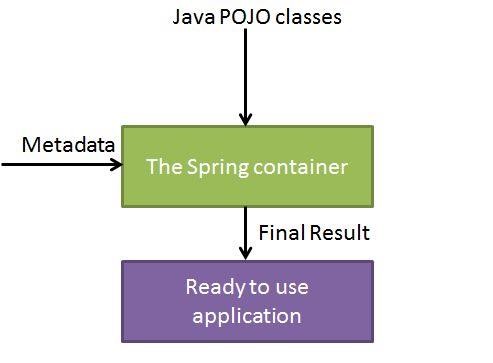


Fig. 2. Spring IoC Container

Spring提供以下两种不同类型的容器。

## A. Spring BeanFactory容器

这是最简单的容器，提供对DI的基本支持。BeanFactory和相关接口（如BeanFactoryAware、InitializingBean、DisposableBean）仍然存在于Spring中，以便与大量集成了Spring的第三方框架保持向后兼容。

***B. Spring ApplicationContext容器***

这个容器增加了更多面向企业的功能，例如从属性文件中解析文本消息的能力，以及向感兴趣的事件监听器发布应用程序事件的能力。这个容器由 org.springframework.context.ApplicationContext接口定义。

ApplicationContext 容器包括了BeanFactory容器的所有功能，因此通常推荐使用ApplicationContext而不是BeanFactory。BeanFactory仍然可以用于轻量级应用程序，比如移动设备或基于小程序的应用程序，这些场景对数据量和速度有较高的要求。

## C. Beans

构成你应用程序骨干并由Spring IoC容器管理的对象称为bean。bean是由Spring IoC容器实例化、组装和管理的对象。这些bean是通过你提供给容器的配置元数据创建的，例如，以XML <bean/> 定义的形式。

## D.Spring配置元数据

Spring IoC容器与配置元数据的实际编写格式完全解耦。有以下三种重要方法可以向Spring容器提供配置元数据：

* 基于XML的配置文件
* 基于注释的配置
* 基于Java的配置

# 四、依赖注入（DI）

Spring最具代表性的技术是控制反转的依赖注入（DI）。控制反转是一个通用概念，它可以通过许多不同的方法来表达，而依赖注入只是控制反转的一个具体例子。

编写复杂的Java应用程序时，应用程序类应尽可能独立于其他Java类，以增加复用这些类的可能性，并在进行单元测试时独立于其他类对它们进行测试。依赖注入有助于将这些类粘合在一起，同时保持它们的独立性。依赖注入到底是什么？让我们分别看看这两个词。这里的依赖部分，转化为两个类之间的关系。例如，类A依赖于类B。现在，让我们看看第二部分，注入。这一切意味着类B将被Ioc注入到类A中。

依赖注入可以通过将参数传递给构造函数的方式进行。假设您有一个包含文本编辑器组件的应用程序，并且您想要提供拼写检查。您的标准代码将如下所示：

public class TextEditor{

privateSpellCheckerspellChecker;

publicTextEditor() {

spellChecker = new SpellChecker();

}

}

我们在这里所作的就是在TextEditor和SpellChecker之间创建依赖关系。在控制反转场景中，我们应该这样做：

public class TextEditor{

privateSpellCheckerspellChecker; publicTextEditor(SpellCheckerspellChecker) {

this.spellChecker = spellChecker;

}

}

在这里，TextEditor不必担心SpellChecker的实现。SpellChecker将被独立实现，并在TextEditor实例化时提供给它，这整个过程由Spring框架控制。我们将控制权从TextEditor中移除，并放在其他地方（即XML配置文件），通过类构造函数将依赖关系（即SpellChecker类）注入到TextEditor类中。因此，控制流通过依赖注入（DI）被“反转”，因为你有效地将依赖关系委托给了某个外部系统。

注入依赖项的第二种方法就是通过TextEditor类的Setter方法，我们将创建SpellChecker实例，并且该实例将用于调用setter方法来初始化TextEditor的属性。

依赖注入有几个重要的好处。例如：

* 由于组件不需要再运行时查找协作对象，因此编写和维护起来更简单。在Spring的Ioc实现中，组件将通过公开JavaEban的setter方法或构造函数参数来表达它们对其他组件的依赖。而EJB的等效实现是通过JNDI查找，这需要开发人员编写假设环境的代码。
* 出于同样的原因，应用程序代码更容易测试。例如，JavaBean属性简单、核心Java易于测试 ：只需要编写创建对象并设置相关属性的JUnit测试方法。
* 良好的Ioc实现保留了强类型。如果您需要使用通用工厂来查找协作对象，则必须将结果转换为所需的类型。这不是一个大问题，但它不够优雅。使用Ioc，您可以在代码中表达强类型依赖关系，而框架负责类型转换。这意味着当框架配置应用程序时，类型不匹配将作为错误引发；您不必担心代码中的类转换异常。
* 依赖关系是明确的。例如，如果一个应用程序类尝试在实例化时加载属性文件或连接到数据库，那么环境假设可能在不阅读代码的情况下并不明显（这会使测试复杂化并降低部署的灵活性）。通过依赖注入方法，依赖关系是明确的，并且在构造函数或JavaBean属性中显而易见。
* 大多数业务对象不依赖于IoC容器的API。这使得使用遗留代码变得容易，并且无论在IoC容器内部还是外部使用对象都很方便。例如，Spring用户经常将Jakarta Commons DBCP DataSource配置为Spring bean：无需编写任何自定义代码即可实现这一点。我们说IoC容器是非侵入性的：使用它不会使你的代码依赖于它的API。几乎任何POJO（Plain Old Java Object）都可以成为Spring bean工厂中的组件。现有的JavaBeans或具有多参数构造函数的对象特别适合，但Spring还提供了独特的支持，可以通过静态工厂方法甚至是由IoC容器管理的其他对象的方法来实例化对象。

Dependency Injection is unlike traditional container architectures, such as EJB, in this minimization of dependency of application code on container. This means that your business objects can potentially be run in different Dependency Injection frameworks - or outside any framework without code changes.

Dependency Injection is not a new concept, although it's only recently made prime time in the J2EE community. There are alternative DI containers: notably, PicoContainer and HiveMind. PicoContainer is particularly lightweight and emphasizes the expression of dependencies through constructors rather than JavaBean properties. It does not use metadata outside Java code, which limits its functionality in comparison with Spring. HiveMind is conceptually more similar to Spring (also aiming at more than just IoC), although it lacks the comprehensive scope of the Spring project or the same scale of user community. EJB 3.0 will provide a basic DI capability as well.

# V. ASPECT ORIENTED PROGRAMMING (AOP)

One of the key components of Spring is the Aspect oriented programming (AOP) framework. The functions that span multiple points of an application are called crosscutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic.

There are various common good examples of aspects including logging, declarative transactions, security, and caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Whereas DI helps you decouple your application objects from each other, AOP helps you decouple cross-cutting concerns from the objects that they affect. The AOP module of Spring Framework provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Spring AOP module provides interceptors to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.[2]

## A. AOP Concepts

* Aspect: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in J2EE applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation

(the @AspectJstyle).

* 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: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.
* 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). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJpointcut expression language by default.
* 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 advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
* Target object: object being advised by one or more aspects. Also referred to as the advisedobject. Since Spring AOP is implemented using runtime proxies, this object will always be aproxied object.
* AOP proxy: An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
* Weaving: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

# VI. SPRING JDBC FRAMEWORK

While working with database using plain old JDBC, it becomes cumbersome to write unnecessary code to handle exceptions, opening and closing database connections etc. But Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.

So what you have to do is just define connection parameters and specify the SQL statement to be executed and do the required work for each iteration while fetching data from the database.

Spring JDBC provides several approaches and correspondingly different classes to interface with the database. I'm going to take classic and the most popular approach which makes use of JdbcTemplate class of the framework. This is the central framework class that manages all the database communication and exception handling. The JdbcTemplateclass executes SQL queries, updates statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the JdbcTemplate class are thread safe once configured. So you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs.A common practice when using the JdbcTemplate class is to configure a DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the

JdbcTemplate is created in the setter for the DataSource.[2]

## A. Data Access Object (DAO)

DAO stands for data access object which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them. The Data Access Object (DAO) support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA or JDO in a consistent way.

## B. Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of and RDBMS oriented enterprise applications to ensure data integrity and consistency.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. The Spring's transaction support aims to provide an alternative to EJB(Enterprise Java Beans) transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs require an application server, but Spring transaction management can be implemented without a need of application server.

* Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system. Localtransaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.
* Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Spring supports two types of transaction management:

* Programmatic transaction management: This means that you have managed the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
* Declarative transaction management: This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

# VII. O/R MAPPING INTEGRATION

Of course often you want to use O/R (Object Relational) mapping, rather than use relational data access. Your overall application framework must support this also. Thus Spring integrates out of the box with Hibernate (versions 2 and 3), JDO (versions 1 and 2), TopLink and other ORM products. Its data access architecture allows it to integrate with *any* underlying data access technology. Spring and Hibernate are a particularly popular combination.

Why would you use an ORM product plus Spring, instead of the ORM product directly? Spring adds significant value in the following areas:

* Session management. Spring offers efficient, easy, and safe handling of units of work such as Hibernate or TopLink Sessions. Related code using the ORM tool alone generally needs to use the same "Session" object for efficiency and proper transaction handling. Spring can transparently create and bind a session to the current thread, using either a declarative, AOP method interceptor approach, or by using an explicit, "template" wrapper class at the Java code level. Thus Spring solves many of the usage issues that affect many users of ORM technology.
* Resource management. Spring application contexts can handle the location and configuration of Hibernate SessionFactories, JDBC datasources, and other related resources. This makes these values easy to manage and change.
* Integrated transaction management. Spring allows you to wrap your ORM code with either a declarative, AOP method interceptor, or an explicit 'template' wrapper class at the Java code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc.) in case of exceptions is taken care of. As we discuss later, you also get the benefit of being able to use and swap various transaction managers, without your ORM-related code being affected. As an added benefit, JDBC-related code can fully integrate transactionally with ORM code, in the case of most supported ORM tools. This is useful for handling functionality not amenable to ORM.
* Exception wrapping. Spring can wrap exceptions from the ORM layer, converting them from proprietary (possibly checked) exceptions, to a set of abstracted runtime exceptions. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. Remember that JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.
* To avoid vendor lock-in. ORM solutions have different performance other characteristics, and there is no perfect one size fits all solution. Alternatively, you may find that certain functionality is just not suited to an implementation using your ORM tool. Thus it makes sense to decouple your architecture from the toolspecific implementations of your data access object interfaces. If you may ever need to switch to another implementation for reasons of functionality, performance, or any other concerns, using Spring now can make the eventual switch much easier. Spring's abstraction of your ORM tool's Transactions and Exceptions, along with its IoC approach which allow you to easily swap in mapper/DAO objects implementing data-access functionality, make it easy to isolate all ORM-specific code in one area of your application, without sacrificing any of the power of your ORM tool. The PetClinic sample application shipped with Spring demonstrates the portability benefits that Spring offers, through providing variants that use JDBC, Hibernate, TopLink and Apache OJB to implement the persistence layer.
* Ease of testing. Spring's inversion of control approach makes it easy to swap the implementations and locations of resources such as Hibernate session factories, datasources, transaction managers, and mapper object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.

Above all, Spring facilitates a mix-and-match approach to data access. Despite the claims of some ORM vendors, ORM is *not* the solution to all problems, although it is a valuable productivity win in many cases.

Spring enables a consistent architecture, and transaction strategy, even if you mix and match persistence approaches, even without using JTA.

Abstracting a data access API is not enough; we also need to consider transaction management. JTA is the obvious solution, but it's a cumbersome API to use directly, and as a result many J2EE developers used to feel that EJB CMT is the only rational option for transaction management. Spring has changed that.

Spring's transaction abstraction is unique in that it's not tied to JTA or any other transaction management technology. Spring uses the concept of a transaction strategy that decouples application code from the underlying transaction infrastructure (such as JDBC).

Why should you care about this? Isn't JTA the best answer for all transaction management? If you're writing an application that uses only a single database, you don't need the complexity of JTA. You're not interested in XA transactions or two phase commit. You may not even need a high-end application server that provides these things. But, on the other hand, you don't want to have to rewrite your code should you ever have to work with multiple data sources.

Imagine you decide to avoid the overhead of JTA by using JDBC or Hibernate transactions directly. If you ever need to work with multiple data sources, you'll have to rip out all that transaction management code and replace it with JTA transactions. This isn't very attractive and led most writers on J2EE, to recommend using global JTA transactions exclusively, effectively ruling out using a simple web container such as Tomcat for transactional applications. Using the Spring transaction abstraction, however, you only have to reconfigure Spring to use a JTA, rather than JDBC or Hibernate, transaction strategy and you're done. This is a configuration change, not a code change. Thus, Spring enables you to write applications that can scale down as well as up.

# VIII. SPRING WEB MVC FRAMEWORK

The Spring web MVC framework provides model-viewcontroller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The Model encapsulates the application data and in general they will consist of POJO.
* The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

## A. The Dispatcher Servlet

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the diagram:

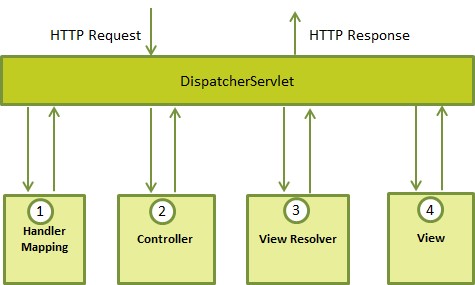


Fig. 3.Request processing workflow of the Spring Web MVC DispatcherServlet Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

* After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.
* The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
* The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.
* Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.[2]

All the above mentioned components i.e.HandlerMapping,

Controller and ViewResolver are parts of

WebApplicationContext which is an extension of the plain ApplicationContext with some extra features necessary for web applications.You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the web.xml file.

Defining a Controller - DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The @Controller annotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method. The @Controller annotation defines the class as a Spring MVC controller.

Creating JSP Views - Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports etc. But most commonly we use JSP templates written with JSTL.

# IX. CONCLUSION

Spring is a powerful framework that solves many common problems in J2EE. Many Spring features are also usable in a wide range of Java environments, beyond classic J2EE.

Spring provides a consistent way of managing business objects and encourages good practices such as programming to interfaces, rather than classes. The architectural basis of Spring is an Inversion of Control container based around the use of JavaBean properties. However, this is only part of the overall picture: Spring is unique in that it uses its IoC container as the basic building block in a comprehensive solution that addresses all architectural tiers.

Spring provides a unique data access abstraction, including a simple and productive JDBC framework that greatly improves productivity and reduces the likelihood of errors. Spring's data access architecture also integrates with TopLink, Hibernate, JDO and other O/R mapping solutions.

Spring also provides a unique transaction management abstraction, which enables a consistent programming model over a variety of underlying transaction technologies, such as JTA or JDBC.

Spring provides an AOP framework written in standard Java, which provides declarative transaction management and other enterprise services to be applied to POJOs or - if you wish - the ability to implement your own custom aspects. This framework is powerful enough to enable many applications to dispense with the complexity of EJB, while enjoying key services traditionally associated with EJB.

# REFERENCES

1. http://www.springsource.org/tutorial
2. http://www.tutorialspoint.com/spring/index.htm
3. http://en.wikipedia.org/wiki/Spring\_Framework
4. http://www.theserverside.com/news/1364527/Introduction-to-theSpring-Framework
5. http://www.theserverside.com/news/1363858/Introduction-to-theSpring-Framework
6. http://www.tutorialspoint.com/spring/spring\_dependency\_injection.ht m
7. Seth Ladd, Darren Davison, Steven Devijver and Colin Yates,

“Expert Spring MVC and Web Flow”

1. Gary Mak , “Spring Recipes”