The Spring Framework: An Open Source Java Platform for Developing Robust Java Applications

Dashrath Mane, Ketaki Chitnis, Namrata Ojha

Abstract*— *The fundamental concepts of Spring Framework is presented in this paper.Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly. The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications.

IndexTerms*— *Aspect Oriented Programming, Dependency Injection, IoC Container, ORM.

I. INTRODUCTION

Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, reusable code. Spring framework is an open source Java platform and it was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments. Spring includes:

- Flexible dependency injection with XML annotation-based configuration styles
- Advanced support for aspect-oriented programming with proxy-based and AspectJ-based variants.
- First-class support for common open source frameworks such as Hibernate and Quartz
- A flexible web framework for building RESTful MVC applications and service endpoints

Spring is modular in design, allowing for incremental adoption of individual parts such as the core container or the JDBC support. While all Spring services are a perfect fit for the Spring core container, many services can also be used in a programmatic fashion outside of the container.

Supported deployment platforms range from standalone applications to Tomcat and Java EE servers such as WebSphere. Spring is also a first-class citizen on major cloud platforms with Java support, e.g. on Heroku, Google App Engine, Amazon Elastic Beanstalk and VMware's Cloud Foundry.[1]

II. SPRING FRAMEWORK ARCHITECTURE

Spring could potentially be a one-stop shop for all your enterprise applications; however, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest.

The Spring Framework provides about 20 modules which can be used based on an application requirement.

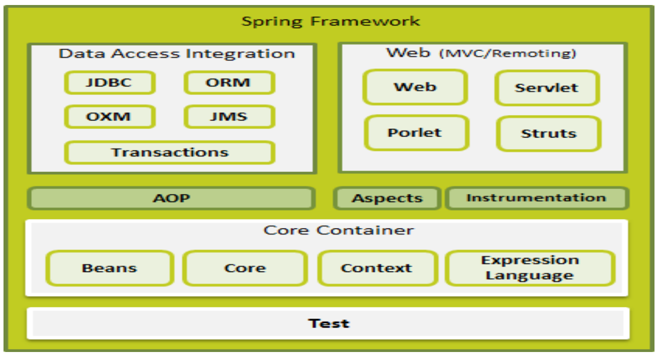

Fig. 1. Spring Framework Architecture

A. Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules whose detail is as follows:

- The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The Bean module provides BeanFactory which is a sophisticated implementation of the factory pattern.
- The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime.

B. Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows:

- The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC related coding.

Manuscript received July, 2013.

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 ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service JMS module contains features for producing and consuming messages.
- The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

C. Web

The Web layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules whose detail is as follows:

- The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The Web-Servlet module contains Spring's model-view-controller (MVC) implementation for web applications.
- The Web-Struts module contains the support classes for integrating a classic Struts web tier within a Spring application.

D. Miscellaneous:

- The AOP module provides aspect-oriented programming implementation allowing you to define method-interceptors and point cuts to cleanly decouple code that implements functionality that should be separated.
- The Aspects module provides integration with AspectJ which is again a powerful and mature aspect oriented programming (AOP) framework.
- The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The Test module supports the testing of Spring components with JUnit or TestNG frameworks.

III. SPRING IOC CONTAINER

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application. These objects are called Spring Beans which we will discuss in next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

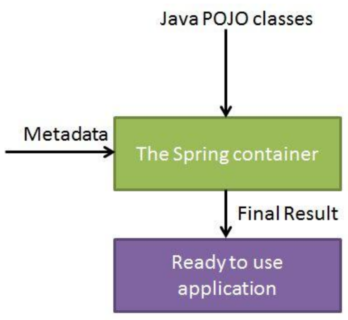

Fig. 2. Spring IoC Container

Spring provides following two distinct types of containers.

A. Spring BeanFactory Container

This is the simplest container providing basic support for DI. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.

B. Spring ApplicationContext Container

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContextinterfac e.

The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.

C. Beans

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.

D.Spring Configuration Metadata

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. There are following three important methods to provide configuration metadata to the Spring Container:

- XML based configuration file.
- Annotation-based configuration
- Java-based configuration

IV. DEPENDENCY INJECTION (DI)

The technology that Spring is most identified with is the Dependency Injection (DI) flavour of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. Dependency Injection helps in gluing these classes together and same time keeping them independent. What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent on class B. Now, let's look at the second part, injection. All this means is that class B will get injected into class A by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods.Consider you have an application which has a text editor component and you want to provide spell checking. Your standard code would look something like this:

```
public class TextEditor {
privateSpellCheckerspellChecker;
publicTextEditor() {
spellChecker = new SpellChecker();
   }
}
```

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario we would instead do something like this:

```
public class TextEditor {
privateSpellCheckerspellChecker;
publicTextEditor(SpellCheckerspellChecker) {
this.spellChecker = spellChecker;
   }
}
```

Here TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to TextEditor at the time of TextEditor instantiation and this entire procedure is controlled by the Spring Framework. We have removed the total control from TextEditor and kept it somewhere else (ie. XML configuration file) and the dependency (ie. class SpellChecker) is being injected into the class TextEditor through a Class Constructor. Thus flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

Second method of injecting dependency is through Setter Methods of TextEditor class where we will create SpellChecker instance and this instance will be used to call setter methods to initialize TextEditor's properties.

Dependency Injection has several important benefits. For example:

- Because components don't need to look up collaborators at runtime, they're much simpler to write and maintain. In Spring's version of IoC, components express their dependency on other components via exposing JavaBean setter methods or through constructor arguments. The EJB equivalent would be a JNDI lookup, which requires the developer to write code that makes environmental assumptions.
- For the same reasons, application code is much easier to test. For example, JavaBean properties are simple, core Java and easy to test: simply write a self-contained JUnit test method that creates the object and sets the relevant properties.
- A good IoC implementation preserves strong typing. If you need to use a generic factory to look up collaborators, you have to cast the results to the desired type. This isn't a major problem, but it is inelegant. With IoC you express strongly typed dependencies in your code and the framework is responsible for type casts. This means that type mismatches will be raised as errors when the framework configures the application; you don't have to worry about class cast exceptions in your code.
- Dependencies are explicit. For example, if an application class tries to load a properties file or connect to a database on instantiation, the environmental assumptions may not be obvious without reading the code (complicating testing and reducing deployment flexibility). With a Dependency Injection approach, dependencies are explicit, and evident in constructor or JavaBean properties.
- Most business objects don't depend on IoC container APIs. This makes it easy to use legacy code, and easy to use objects either inside or outside the IoC container. For example, Spring users often configure the Jakarta Commons DBCP DataSource as a Spring bean: there's no need to write any custom code to do this. We say that an IoC container isn't invasive: using it won't invade your code with dependency on its APIs. Almost any POJO can become a component in a Spring bean factory. Existing JavaBeans or objects with multi-argument constructors work particularly well, but Spring also provides unique support for instantiating objects from static factory methods or even methods on other objects managed by the IoC container.

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 cross-cutting 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 tool-specific 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-view-controller 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:

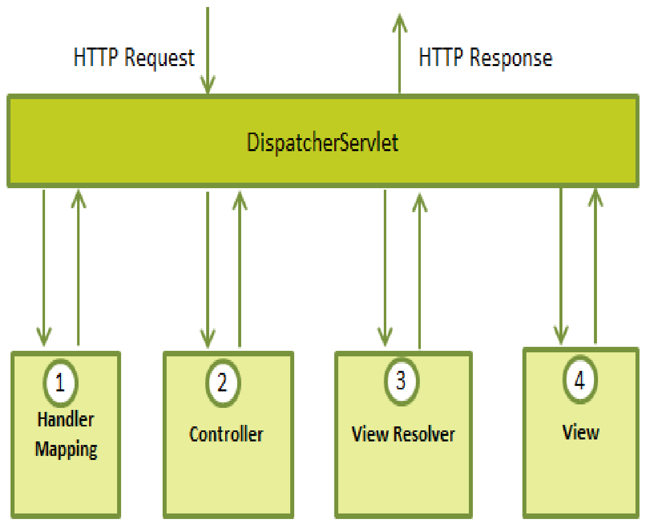

Fig. 3.Request processing workflow of the Spring Web MVC DispatcherServlet

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-the-Spring-Framework
[5] http://www.theserverside.com/news/1363858/Introduction-to-the-Spring-Framework
[6] http://www.tutorialspoint.com/spring/spring_dependency_injection.htm
[7] Seth Ladd, Darren Davison, Steven Devijver and Colin Yates, "Expert Spring MVC and Web Flow"
[8] Gary Mak , "Spring Recipes"