**Spring Framework**

Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

**Advantages of Spring Framework**

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JDBC Template; you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it saves a lot of JDBC code.

2) Loose Coupling

The spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

**Inversion Of Control (IOC) and Dependency Injection**

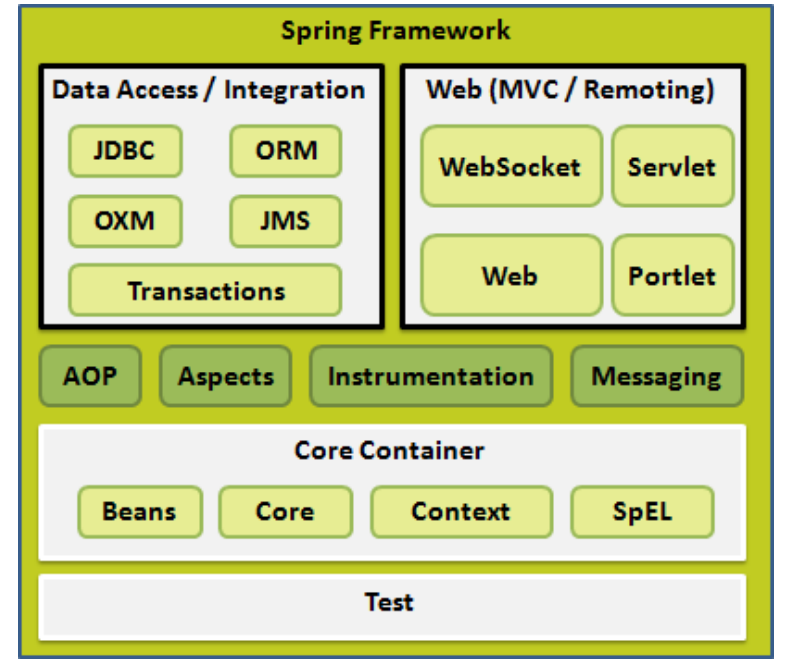
The technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control.

Here the dependency part translates into an association between two classes. For example, class B is dependent of class A. Now, let’s look at the second part, injection. This entire means is, class A will get injected into class B by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework.

**Advantage of Dependency Injection**

* makes the code loosely coupled so easy to maintain
* makes the code easy to test

**Architecture of Spring Framework**

**Core Container**

1. The Core module provides the fundamental parts of the framework, including the **IoC** and Dependency Injection features.
2. The Bean module provides **BeanFactory**, which is a sophisticated implementation of the factory pattern.
3. 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.
4. The SpEL module provides a powerful expression language for querying and manipulating an object graph at runtime.

**Data Access/Integration**

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

1. The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
2. The **ORM** module provides integration layers for popular object-relational mappingAPIs, including JDO, Hibernate, and iBatis.
3. The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, and XStream.
4. The Java Messaging Service (JMS) module contains features for producing and consuming messages.
5. The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

**Web**

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules

1. 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.
2. The Web-MVC module contains Spring’s Model-View-Controller (MVC) implementation for web application.
3. The Web-Socket module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
4. The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

**Note: -**

Spring Framework provides 2 types of container.

1. Spring **BeanFactory** container
2. Spring **ApplicationContext** container
3. Spring **BeanFactory** container

It is the simplest container providing the basic support for DI.

It is defined by the **org.springframework.beans.factory.BeanFacto**ryinterface.

**XMLBeanFactory** is a class and this container reads the configuration metadata from an XML file and uses it to create a fully configured application.

**Constructor of XMLBeanFactory class:-**

XMLBeanFactory(ClassPathResource);

**Eg:-**

XMLBeanFactory f=new XMLBeanFactory(new ClassPathResource(“beans.xml”));

f.getBean(beanName); to get the required bean object.

1. 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.

It is defined by the **org.springframework.context.ApplicationContext** interface. The most commonly used ApplicationContext implementations are,

FileSystemXmlApplicationContext, WebXmlApplicationContext and

ClassPathXmlApplicationContext (It is easy and best one)

**Eg:-**

ApplicationContext ctx=new ClassPathXmlApplicationContext (“beans.xml”);

ctx.getBean(beanName);

**Spring - Bean Definition**

The objects that form the backbone of our application and that are managed by the Spring IoC container are called **beans**.

A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container.

Bean definition contains the information called **configuration metadata (**bean tag in xml file), which is needed for the container to know the following:

* How to create a bean
* Bean's lifecycle details
* Bean's dependencies

**Bean Properties:-**

Class 🡪 to specify the bean class to create the bean.

Id (or) name 🡪 to specify bean identifier.

Scope 🡪 To specify the scope of the bean. 2 Scopes available.

Singleton 🡪 single instance per Spring IoC container (default).

Prototype 🡪 single bean definition to have any number of object instances.

Constructor-arg 🡪 To inject the dependency

Eg:-

<bean id=”e1” class=”priyaa.Employee”>

<property name=”empid” value=”10001”/>

<property name=”name” value=”Sasi”/>

</bean>

Here, we are going to create a simple application of spring framework using eclipse IDE. Let's see the simple steps to create the spring application in Eclipse IDE.

1. create the java project
2. add spring jar files
3. create a java class
4. create the xml file to initialize the bean and provide the values
5. create a java class (with main function)

**Steps to create spring application in Eclipse IDE**

Let's see the 5 steps to create the first spring application using eclipse IDE.

1) Create the Java Project

Go to File menu - New - project - Java Project. Write the project name e.g. firstspring - Finish. Now the java project is created.

2) Add spring jar files

There are mainly three jar files required to run this application.

org.springframework.core-3.0.1.RELEASE-A.jar

com.springsource.org.apache.commons.logging-1.1.1.jar

org.springframework.beans-3.0.1.RELEASE-A.jar

To load the jar files in eclipse IDE, Right click on your project - Build Path - Add external archives - select all the required jar files - finish..

3) Create Java class

To create the java class, Right click on src - New - class - Write the class name e.g. Student - finish.

4) Create the xml file

To create the xml file click on src - new - file - give the file name such as beans.xml - finish. Open the beans.xml file, and write the following code:

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="studentbean" class="madurai.Student">

<property name="name" value="Harini.S"></property>

</bean>

</beans>

Here the bean element is used to define the bean for the given class. The property sub-element of bean specifies the property of the Student class named name. The value specified in the property element will be set in the Student class object by the IOC container.

5) Create the main class

Create the java class e.g. Test. Here we are getting the object of Student class from the IOC container using the getBean() method of BeanFactory. Let's see the code of test class.

package priyaa;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

public class Test {

public static void main(String[] args) {

Resource resource=new ClassPathResource("beans.xml");

BeanFactory factory=new XmlBeanFactory(resource);

Student student = (Student)factory.getBean("studentbean");

student.displayInfo();

}

}

**Spring – Dependency Injection**

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 unit testing.

Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

Dependency injection can be done through either **constructor** or through **setter methods**.

**Constructor-based Dependency Injection**

Eg:-

<bean id=”b1” class=””>

<constructor-arg ref=””/>

<constructor-arg type=”int” value=”1604”/>

<constructor-arg index=”0” value=”Madurai”/>

</bean>

**Setter-based Dependency Injection**

<bean id=”b1” class=””>

<property name=”” ref=””/>

<property name=”city” value=”Madurai”/>

</bean>

**Spring – Beans Auto Wiring**

The Spring container can auto wire relationships between collaborating beans without using <constructor-arg> and <property> elements, which helps cut down on the amount of XML configuration .

**Auto wiring ‘byName’**

This mode specifies auto wiring by property name. Spring container looks at the beans on which auto-wire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. If matches are found, it will inject those beans. Otherwise, it will throw exceptions.

Eg:-

<bean id=”b1” class=”” autowire=”byname”>

<property name=”city” value=”Madurai”/>

</bean>

**Auto wiring ‘byType’**

This mode specifies auto wiring by property type. Spring container looks at the beans on which auto wire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in the configuration file. If matches are found, it will inject those beans. Otherwise, it will throw exceptions.

**Spring – Annotation Based Configuration**

We can configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, we can use the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

**Beans.XML file:-**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring -context-3.0.xsd">

<context:annotation-config/>

<!-- bean definitions go here -->

</beans>

Note:-

Once <context:annotation-config/> is configured, we can start annotating our code to

indicate that Spring should automatically wire values into properties, methods, and constructors.

@Required 🡪 This annotation applies to bean property setter methods.

@Autowired 🡪 This annotation can apply to bean property setter methods, non-setter methods, constructor and properties

@Qualifier 🡪 This annotation along with @Autowired can be used to remove the confusion by specifying which exact bean will be wired.

**Spring – Java Based Configuration**

Java-based configuration option enables you to write most of your Spring configuration without XML

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.

The **@Bean** annotation tells spring that a method annotated with @Bean will return an object that should be registered as a bean in the spring application context.

Eg:-

package madurai;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

package madurai;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

**The above code equivalent to following XML configuration:-**

<beans>

<bean id="helloWorld" class="madurai.HelloWorld" />

</beans>

package madurai;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld h1 = ctx.getBean(HelloWorld.class);

h1.setMessage("Hello World!");

h1.getMessage();

}

}

**Spring - JdbcTemplate**

It is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC API, but eliminates a lot of problems of JDBC API.

**Problems of JDBC API:-**

\* We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.

\* We need to perform exception handling code on the database logic.

\* We need to handle transaction.

\* Repetition of all these codes from one database to another database logic is a time consuming task.

**Spring JDBC Approaches**

Spring framework provides following approaches for JDBC database access:

1. JdbcTemplate
2. NamedParameterJdbcTemplate
3. SimpleJdbcTemplate
4. SimpleJdbcInsert and SimpleJdbcCall

**JdbcTemplate class**

It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

**Methods of JdbcTemplate class**

1. Public int update(String query);

To insert, update and delete records.

1. Public int update(String query, Object …args);

To insert, update and delete records using PreparedStatement using given arguments.

1. Public boolean execute(String query);

For executing DDL command.

1. Public int queryForInt(String query);
2. Public int queryForInt(String query, Object …args);
3. Public long queryForLong(String query);
4. Public long queryForLong(String query, Object …args);
5. Public <T> queryForObject(String query, RowMapper<T>);
6. Public <T> queryForObject(String query, Object[] args, RowMapper<T>);
7. Public List<T> query (String query, RowMapper<T>)
8. Public List<T> query (String query, Object[] args, RowMapper<T>);
9. Public List<T> query (String query, RowMapper<T>,Object ...args);

**Initialization for DataSource and JDBCTemplate in XML file:-**

<bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource" >

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

<property name="url" value="jdbc:mysql://localhost:3306/sasi"/>

<property name="username" value="root"/>

<property name="password" value=" "/>

</bean>

<bean id=”jdbc” class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="ds" />

</bean>

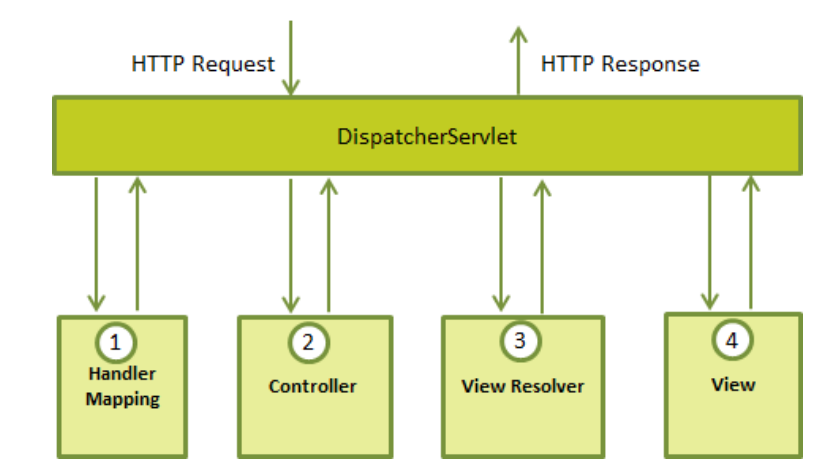
**Spring MVC Framework**

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 an appropriate model and passes it to the view for rendering.

The Spring Web MVC framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.



Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

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