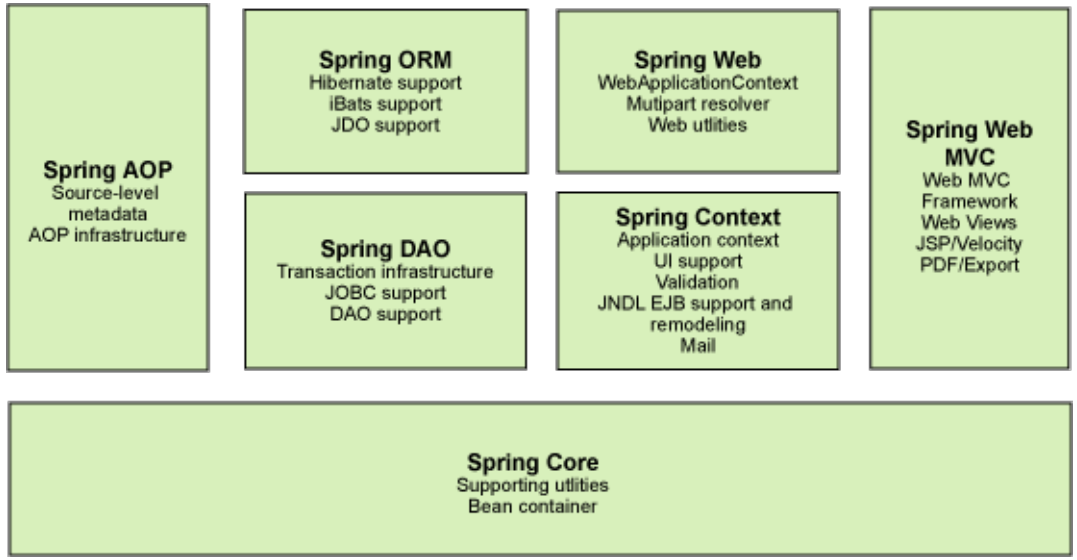
Spring

关注微信公众号
享终身免费学习

1. Overview

Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly. Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003. This tutorial has been written based on Spring Framework version 4.1.6 released in Mar 2015.



2. Why to Learn Spring?

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, and reusable code.

Spring framework is an open source Java platform. 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 promotes good programming practices by enabling a POJO-based programming model.

3. Applications of Spring

Following is the list of few of the great benefits of using Spring Framework −

- **POJO Based** - Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- **Modular** - Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
- **Integration with existing frameworks** - Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- **Testablity** - Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
- **Web MVC** - Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- **Central Exception Handling** - Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- **Lightweight** - Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.

- **Transaction management** - Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

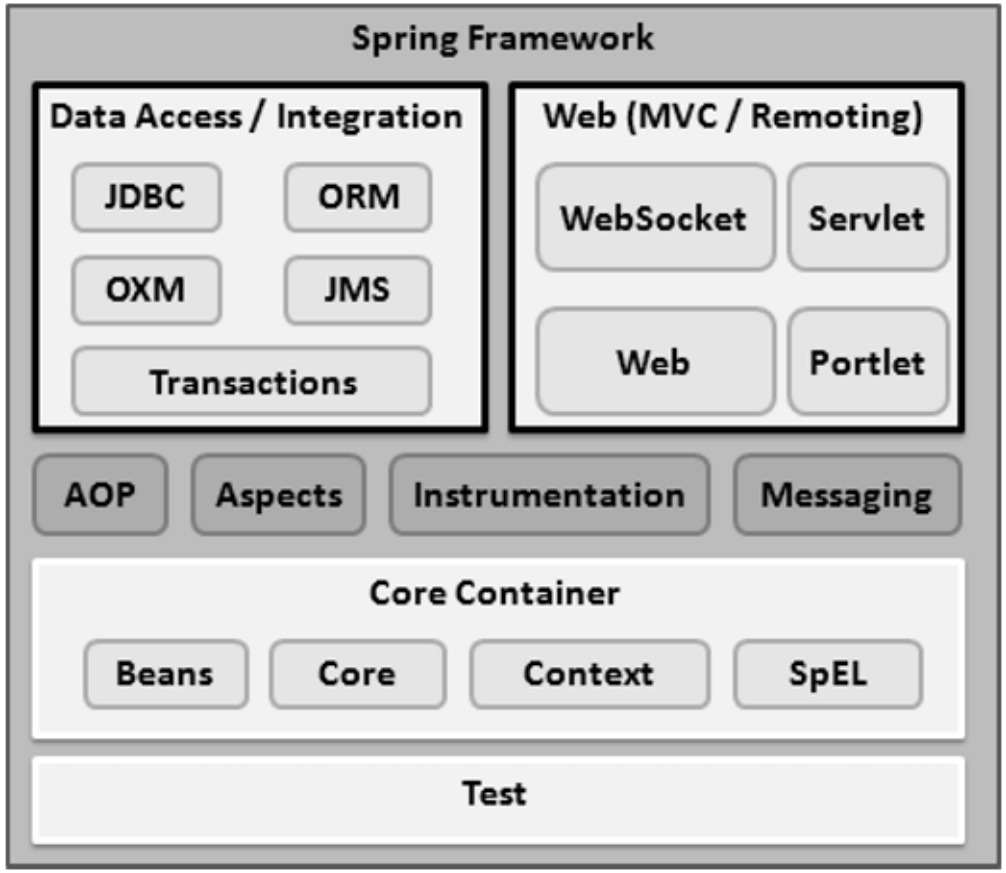
4. Benefits of Using the Spring Framework

Following is the list of few of the great benefits of using Spring Framework −

- Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
- Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
- Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
- Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

5. Architecture

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



5.1. Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are 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 **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

5.2. 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 for tedious JDBC related coding.
- 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.

5.3. Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are 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-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

5.4. Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts 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 AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

6. Hello World Example

6.1. Step 1 - Create Java Project

The first step is to create a simple Java Project

6.2. Step 2 - Add Required Libraries

- commons-logging-1.1.1
- spring-aop-4.1.6.RELEASE
- spring-aspects-4.1.6.RELEASE
- spring-beans-4.1.6.RELEASE
- spring-context-4.1.6.RELEASE
- spring-context-support-4.1.6.RELEASE
- spring-core-4.1.6.RELEASE
- spring-expression-4.1.6.RELEASE
- spring-instrument-4.1.6.RELEASE
- spring-instrument-tomcat-4.1.6.RELEASE
- spring-jdbc-4.1.6.RELEASE
- spring-jms-4.1.6.RELEASE
- spring-messaging-4.1.6.RELEASE
- spring-orm-4.1.6.RELEASE
- spring-oxm-4.1.6.RELEASE
- spring-test-4.1.6.RELEASE
- spring-tx-4.1.6.RELEASE
- spring-web-4.1.6.RELEASE
- spring-webmvc-4.1.6.RELEASE
- spring-webmvc-portlet-4.1.6.RELEASE

- spring-websocket-4.1.6.RELEASE

6.3. Step 3 - Create Source Files

HelloWorld.java

```
package com.tutorial;

public class HelloWorld {
  private String message;

  public void setMessage(String message){
    this.message  = message;
  }
  public void getMessage(){
    System.out.println("Your Message : " + message);
  }
}
```

MainApp.java

Hello World Example


```
package com.tutorial;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
    HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
    obj.getMessage();
  }
}
```

Following two important points are to be noted about the main program −

- The first step is to create an application context where we used framework API **ClassPathXmlApplicationContext()**. This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.
- The second step is used to get the required bean using **getBean()** method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

6.4. Step 4 - Create Bean Configuration 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"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id = "helloWorld" class = "com.tutorial.HelloWorld">
      <property name = "message" value = "Hello World!"/>
   </bean>

</beans>
```

When Spring application gets loaded into the memory, Framework makes use of the above configuration file to create all the beans defined and assigns them a unique ID as defined in `<bean>` tag. You can use `<property>` tag to pass the values of different variables used at the time of object creation.

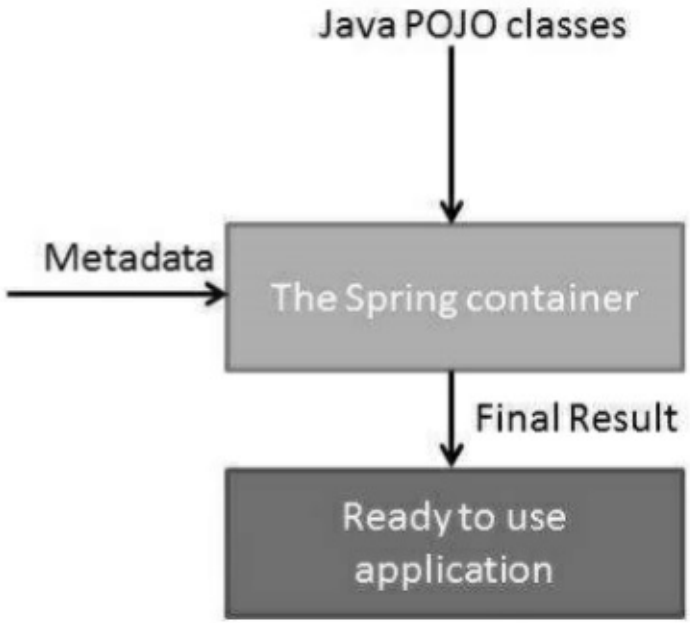
6.5. Step 5 - Running the Program

```
Your Message : Hello World!
```

7. IoC Containers

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

- to instantiate the application class
- to configure the object
- to assemble the dependencies between the objects



There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

7.1. Spring BeanFactory Container

This is the simplest container providing the basic support for DI and is defined by the *org.springframework.beans.factory.BeanFactory* interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

7.2. 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.ApplicationContext* interface.

8. Bean Definition

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.

Sr.No.	Properties & Description
1	**class**This attribute is mandatory and specifies the bean class to be used to create the bean.
2	**name**This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
3	**scope**This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter.
4	**constructor-arg**This is used to inject the dependencies and will be discussed in subsequent chapters.
5	**properties**This is used to inject the dependencies and will be discussed in subsequent chapters.
6	**autowiring mode**This is used to inject the dependencies and will be discussed in subsequent chapters.
7	**lazy-initialization mode**A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup.
8	**initialization method**A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter.
9	**destruction method**A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter.

9. Spring Configuration Metadata

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

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

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

<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- A simple bean definition -->
  <bean id = "..." class = "...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- A bean definition with lazy init set on -->
  <bean id = "..." class = "..." lazy-init = "true">
    <!-- collaborators and configuration for this bean go here -->
  </bean>
```

```
<!-- A bean definition with initialization method -->
<bean id = "..." class = "..." init-method = "...">
   <!-- collaborators and configuration for this bean go here -->
</bean>

<!-- A bean definition with destruction method -->
<bean id = "..." class = "..." destroy-method = "...">
   <!-- collaborators and configuration for this bean go here -->
</bean>

<!-- more bean definitions go here -->

</beans>
```

10. Bean Scopes

When defining a you have the option of declaring a scope for that bean. For example, to force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**. Similarly, if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton**.

The Spring Framework supports the following five scopes, three of which are available only if you use a web-aware ApplicationContext.

Sr.No.	Scope & Description
1	**singleton**This scopes the bean definition to a single instance per Spring IoC container (default).
2	**prototype**This scopes a single bean definition to have any number of object instances.
3	**request**This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
4	**session**This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
5	**global-session**This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

11. Bean Life Cycle

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there are lists of the activities that take place behind the scene between the time of bean Instantiation and its destruction, this chapter will discuss only two important bean life cycle callback methods, which are required at the time of bean initialization and its destruction.

To define setup and teardown for a bean, we simply declare the with **initmethod** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroymethod specifies a method that is called just before a bean is removed from the container.

12. Beans Auto-Wiring

You have learnt how to declare beans using the `<bean>` element and inject `<bean>` using `<constructor-arg>` and `<property>` elements in XML configuration file.

The Spring container can **autowire** relationships between collaborating beans without using `<constructor-arg>` and `<property>` elements, which helps cut down on the amount of XML configuration you write for a big Spring-based application.

13. Annotation Based Configuration

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration 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>
```

Once `<context:annotation-config/>` is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. Let us look at a few important annotations to understand how they work −

- `@Required` - The @Required annotation applies to bean property setter methods.
- `@Autowired` - The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.

- `@Qualifier` - The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
- JSR-250 Annotations - Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations.

14. Dependency Injection

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled. To understand the DI better, Let’s understand the Dependency Lookup (DL) first:

14.1. Dependency Lookup

The Dependency Lookup is an approach where we get the resource after demand. There can be various ways to get the resource for example:

```
A obj = new AImpl();
```

In such way, we get the resource(instance of A class) directly by new keyword. Another way is factory method:

```
A obj = A.getA();
```

This way, we get the resource (instance of A class) by calling the static factory method getA().

Alternatively, we can get the resource by JNDI (Java Naming Directory Interface) as:

```
Context ctx = new InitialContext();
Context environmentCtx = (Context) ctx.lookup("java:comp/env");
A obj = (A)environmentCtx.lookup("A");
```

There can be various ways to get the resource to obtain the resource. Let's see the problem in this approach.

Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

- **tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
- **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

14.2. Dependency Injection

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

```
class Employee{
   Address address;

   Employee(Address address){
      this.address=address;
   }
   public void setAddress(Address address){
      this.address=address;
   }
}
```

In such case, instance of Address class is provided by external souce such as XML file either by constructor or setter method.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

- By Constructor
- By Setter method

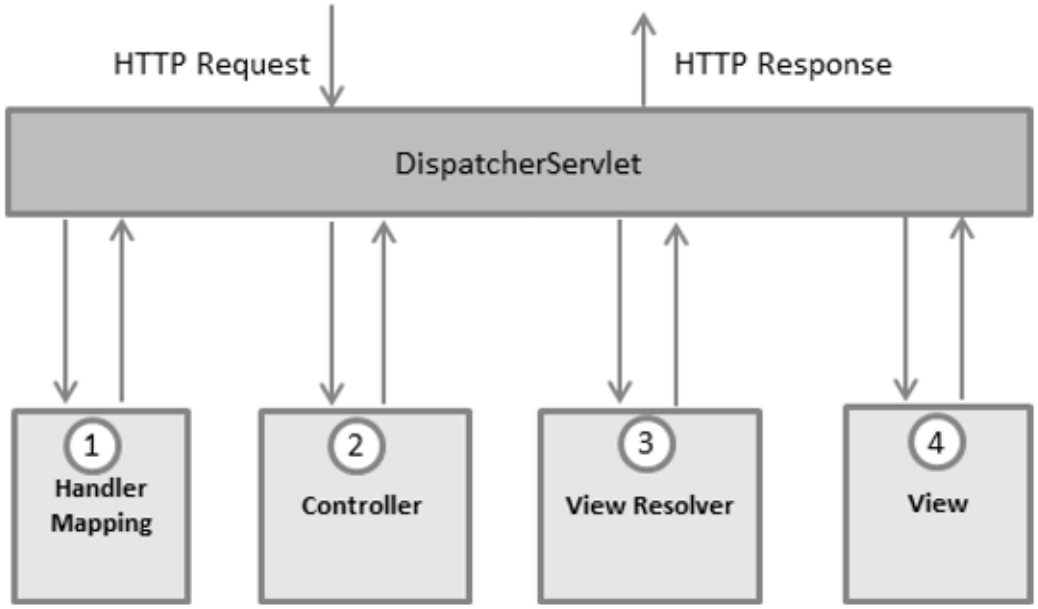
15. MVC Framework

The Spring Web MVC framework provides Model-View-Controller (MVC) 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 an appropriate model and passes it to the view for rendering.

15.1. The DispatcherServlet

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 following diagram −



图一、 Spring 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.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* w which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

15.2. Spring MVC Hello World Example

The following example shows how to write a simple web-based Hello World application using Spring MVC framework. To start with it, let us have a working Eclipse IDE in place and take the following steps to develop a Dynamic Web Application using Spring Web Framework −

Steps	Description
1	Create a *Dynamic Web Project* with a name *HelloWeb* and create a package *com.tutorial* under the *src* folder in the created project.
2	Drag and drop below mentioned Spring and other libraries into the folder *WebContent/WEB-INF/lib*.
3	Create a Java class *HelloController* under the *com.tutorial* package.
4	Create Spring configuration files *web.xml* and *HelloWeb-servlet.xml* under the *WebContent/WEB-INF* folder.
5	Create a sub-folder with a name *jsp* under the *WebContent/WEB-INF* folder. Create a view file *hello.jsp* under this sub-folder.
6	The final step is to create the content of all the source and configuration files and export the application as explained below.

Here is the content of **HelloController.java** file

```
package com.tutorial;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.ui.ModelMap;

@Controller
@RequestMapping("/hello")
public class HelloController {
   @RequestMapping(method = RequestMethod.GET)public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

Following is the content of Spring Web configuration file **web.xml**

```
<web-app id = "WebApp_ID" version = "2.4"
   xmlns = "http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

   <display-name>Spring MVC Application</display-name>

   <servlet>
      <servlet-name>HelloWeb</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>HelloWeb</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>

</web-app>
```

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

```
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:context = "http://www.springframework.org/schema/context"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  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:component-scan base-package = "com.tutorial" />

  <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value = "/WEB-INF/jsp/" />
    <property name = "suffix" value = ".jsp" />
  </bean>

</beans>
```

Following is the content of Spring view file **hello.jsp**

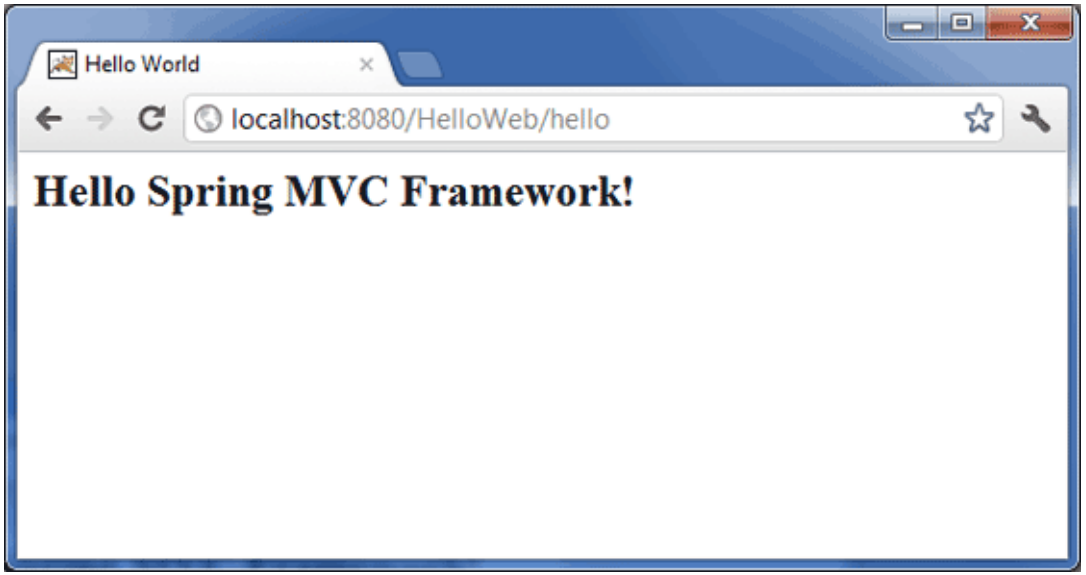
```
<%@ page contentType = "text/html; charset = UTF-8" %>
<html>
  <head>
    <title>Hello World</title>
  </head>

  <body>
    <h2>${message}</h2>
  </body>
</html>
```

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

- commons-logging-x.y.z.jar
- org.springframework.asm-x.y.z.jar
- org.springframework.beans-x.y.z.jar
- org.springframework.context-x.y.z.jar
- org.springframework.core-x.y.z.jar
- org.springframework.expression-x.y.z.jar
- org.springframework.web.servlet-x.y.z.jar
- org.springframework.web-x.y.z.jar
- spring-web.jar

Once you are done creating the source and configuration files, export your application. Right-click on your application and use **Export > WAR File** option and save your **HelloWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Try to access the URL **http://localhost:8080/HelloWeb/hello** and if everything is fine with your Spring Web Application, you should see the following result −



图二、 Spring Web Hello World

You should note that in the given URL, **HelloWeb** is the application name and **hello** is the virtual subfolder which we have mentioned in our controller using @RequestMapping("/hello"). You can use direct root while mapping your URL using **@RequestMapping("/")**. In this case you can access the same page using short URL **http://localhost:8080/HelloWeb/** but it is advised to have different functionalities under different folders.

16. JDBC Framework

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.

16.1. JdbcTemplate Class

The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts 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 *threadsafe* 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 JDBC Template 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.

16.2. Configuring Data Source

Let us create a database table **Student** in our database **TEST**. We assume you are working with MySQL database, if you work with any other database then you can change your DDL and SQL queries accordingly.

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Now we need to supply a DataSource to the JDBC Template so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code as shown in the following code snippet −

```
<bean id = "dataSource"
   class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>
   <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>
   <property name = "username" value = "root"/>
   <property name = "password" value = "password"/>
</bean>
```

16.3. 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 DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

16.4. Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and JDBC Template object.

Querying for an integer

```
String SQL = "select count(*) from Student";
int rowCount = jdbcTemplateObject.queryForInt( SQL );
```

Querying for a long

```
String SQL = "select count(*) from Student";
long rowCount = jdbcTemplateObject.queryForLong( SQL );
```

A simple query using a bind variable

```
String SQL = "select age from Student where id = ?";
int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});
```

Querying for a String

```
String SQL = "select name from Student where id = ?";
String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class);
```

Querying and returning an object

```
String SQL = "select * from Student where id = ?";
Student student = jdbcTemplateObject.queryForObject(
  SQL, new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
  public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
    Student student = new Student();
    student.setID(rs.getInt("id"));
    student.setName(rs.getString("name"));
    student.setAge(rs.getInt("age"));

    return student;
  }
}
```

Querying and returning multiple objects

```
String SQL = "select * from Student";
List<Student> students = jdbcTemplateObject.query(
  SQL, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
  public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
    Student student = new Student();
    student.setID(rs.getInt("id"));
    student.setName(rs.getString("name"));
    student.setAge(rs.getInt("age"));

    return student;
  }
}
```

Inserting a row into the table

```
String SQL = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );
```

Updating a row into the table

```
String SQL = "update Student set name = ? where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );
```

Deleting a row from the table

```
String SQL = "delete Student where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{20} );
```

16.5. Executing DDL Statements

You can use the **execute(..)** method from *jdbcTemplate* to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table −

```
String SQL = "CREATE TABLE Student( " +
  "ID   INT NOT NULL AUTO_INCREMENT, " +
  "NAME VARCHAR(20) NOT NULL, " +
  "AGE  INT NOT NULL, " +
  "PRIMARY KEY (ID));"

jdbcTemplateObject.execute( SQL );
```

16.6. Spring JDBC Framework Examples

To understand the concepts related to Spring JDBC framework with JdbcTemplate class, let us write a simple example, which will implement all the CRUD operations on the following Student table.

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Before proceeding, let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

Steps	Description
1	Create a project with a name *SpringExample* and create a package *com.tutorial* under the **src** folder in the created project.
2	Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter.
3	Add Spring JDBC specific latest libraries **mysql-connector-java.jar**, **org.springframework.jdbc.jar** and **org.springframework.transaction.jar** in the project. You can download required libraries if you do not have them already.
4	Create DAO interface *StudentDAO* and list down all the required methods. Though it is not required and you can directly write *StudentJDBCTemplate* class, but as a good practice, let's do it.
5	Create other required Java classes *Student*, *StudentMapper*, *StudentJDBCTemplate* and *MainApp* under the *com.tutorial* package.
6	Make sure you already created **Student** table in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the give username and password.

Steps	Description
7	Create Beans configuration file *Beans.xml* under the **src** folder.
8	The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Following is the content of the Data Access Object interface file **StudentDAO.java** −

```
package com.tutorial;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
  /**
     * This is the method to be used to initialize
     * database resources ie. connection.
  */
  public void setDataSource(DataSource ds);

  /**
     * This is the method to be used to create
     * a record in the Student table.
  */
  public void create(String name, Integer age);

  /**
     * This is the method to be used to list down
     * a record from the Student table corresponding
```

```
    a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);

   /**
    * This is the method to be used to list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();

   /**
    * This is the method to be used to delete
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public void delete(Integer id);

   /**
    * This is the method to be used to update
    * a record into the Student table.
    */
   public void update(Integer id, Integer age);
}
```

Following is the content of the **Student.java** file

```
package com.tutorial;

public class Student {
  private Integer age;
  private String name;
  private Integer id;

  public void setAge(Integer age) {
    this.age = age;
  }
  public Integer getAge() {
    return age;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void setId(Integer id) {
    this.id = id;
  }
  public Integer getId() {
    return id;
  }
}
```

Following is the content of the **StudentMapper.java** file

```
package com.tutorial;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {
  public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
    Student student = new Student();
    student.setId(rs.getInt("id"));
    student.setName(rs.getString("name"));
    student.setAge(rs.getInt("age"));

    return student;
  }
}
```

Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO.

```
package com.tutorial;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO {
```

```
private DataSource dataSource;
private JdbcTemplate jdbcTemplateObject;

public void setDataSource(DataSource dataSource) {
  this.dataSource = dataSource;
  this.jdbcTemplateObject = new JdbcTemplate(dataSource);
}
public void create(String name, Integer age) {
  String SQL = "insert into Student (name, age) values (?, ?)";
  jdbcTemplateObject.update( SQL, name, age);
  System.out.println("Created Record Name = " + name + " Age = " + age);
  return;
}
public Student getStudent(Integer id) {
  String SQL = "select * from Student where id = ?";
  Student student = jdbcTemplateObject.queryForObject(SQL,
    new Object[]{id}, new StudentMapper());

  return student;
}
public List<Student> listStudents() {
  String SQL = "select * from Student";
  List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
  return students;
}
public void delete(Integer id) {
  String SQL = "delete from Student where id = ?";
```

```
    jdbcTemplateObject.update(SQL, id);
    System.out.println("Deleted Record with ID = " + id );
    return;
  }
  public void update(Integer id, Integer age){
    String SQL = "update Student set age = ? where id = ?";
    jdbcTemplateObject.update(SQL, age, id);
    System.out.println("Updated Record with ID = " + id );
    return;
  }
}
```

Following is the content of the **MainApp.java** file

```
package com.tutorial;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorial.StudentJDBCTemplate;

public class MainApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

    StudentJDBCTemplate studentJDBCTemplate =
```

```
      (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

    System.out.println("------Records Creation--------" );
    studentJDBCTemplate.create("Zara", 11);
    studentJDBCTemplate.create("Nuha", 2);
    studentJDBCTemplate.create("Ayan", 15);

    System.out.println("------Listing Multiple Records--------" );
    List<Student> students = studentJDBCTemplate.listStudents();

    for (Student record : students) {
      System.out.print("ID : " + record.getId() );
      System.out.print(", Name : " + record.getName() );
      System.out.println(", Age : " + record.getAge());
    }

    System.out.println("----Updating Record with ID = 2 -----" );
    studentJDBCTemplate.update(2, 20);

    System.out.println("----Listing Record with ID = 2 -----" );
    Student student = studentJDBCTemplate.getStudent(2);
    System.out.print("ID : " + student.getId() );
    System.out.print(", Name : " + student.getName() );
    System.out.println(", Age : " + student.getAge());
  }
}
```

Following is the configuration file **Beans.xml**

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

   <!-- Initialization for data source -->
   <bean id="dataSource"
      class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>
      <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>
      <property name = "username" value = "root"/>
      <property name = "password" value = "password"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id = "studentJDBCTemplate"
      class = "com.tutorial.StudentJDBCTemplate">
      <property name = "dataSource" ref = "dataSource" />
   </bean>

</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

```
------Records Creation--------
Created Record Name = Zara Age = 11
Created Record Name = Nuha Age = 2
Created Record Name = Ayan Age = 15
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 2
ID : 3, Name : Ayan, Age : 15
----Updating Record with ID = 2 -----
Updated Record with ID = 2
----Listing Record with ID = 2 -----
ID : 2, Name : Nuha, Age : 20
```

You can try and delete the operation yourself, which we have not used in the example, but now you have one working application based on Spring JDBC framework, which you can extend to add sophisticated functionality based on your project requirements. There are other approaches to access the database where you will use **NamedParameterJdbcTemplate** and **SimpleJdbcTemplate** classes, so if you are interested in learning these classes then kindly check the reference manual for Spring Framework.

Thank you

全国统一咨询热线：400-690-6115

北京|上海|广州|深圳|天津|成都|重庆|武汉|济南|青岛|杭州|西安