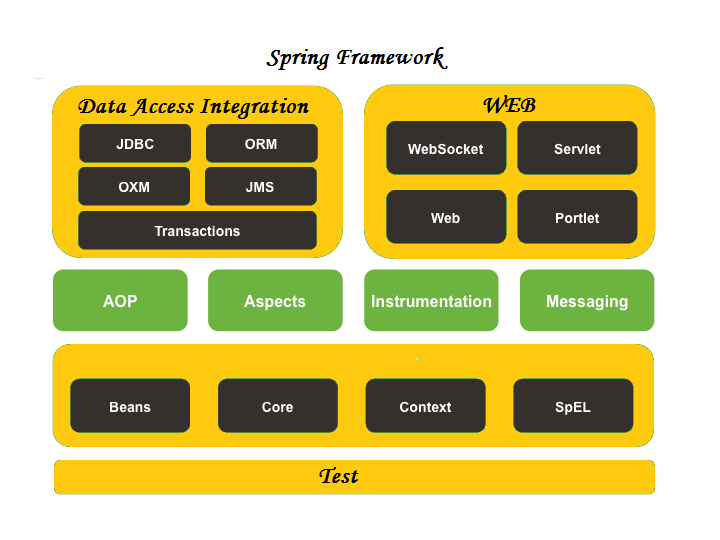
**SPRING MVC**

1. Spring MVC architecture
2. Inversion Of Control and Dependency Injection
3. IOC containers
4. Spring BeanFactory Container
5. Spring ApplicationContext Container
6. Dependency Injection
7. Constructor-based DI
8. Setter-based DI
9. Exercise
10. Code link

**SPRING MVC ARCHITECTURE:**

****

**INVERSION OF CONTROL AND DEPENDENCY INJECTION:**

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain.

class Employee{

Address address;

Employee(){

address=new Address();

}

}

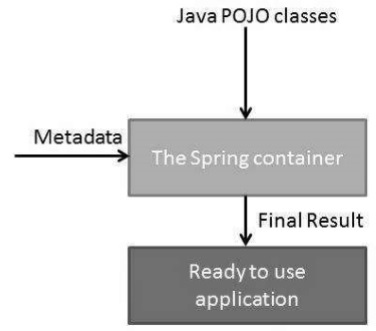
In the above code, dependency between employee and address is high or it is tightly coupled .This coupling can be minimised with the help of IOC . Inversion of control makes the code loosely coupled.

In Spring framework, IOC container is responsible to inject the dependency.

**IOC CONTAINERS:**

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 life cycle from creation till destruction. The Spring container uses DI(Dependency Injection) to manage the components that make up an application. The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code.

The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



#### **Spring ApplicationContext Container:**

The Application Context is Spring's advanced container. Similar to BeanFactory, it can load bean definitions, wire beans together, and dispense beans upon request. Additionally, it 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 *org.springframework.context.ApplicationContext* interface.

The most commonly used ApplicationContext implementations are − (First two implementations are used for loading the definitions of the beans from an XML file).

**1. FileSystemXmlApplicationContext -** need to provide the full path of the XML bean configuration file to the constructor.

**Example: ApplicationContext context = new FileSystemXmlApplicationContext**

**("C:/Users/ZARA/workspace/HelloSpring/src/Beans.xml");**

**2. ClassPathXmlApplicationContext -** do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look like bean configuration XML file in CLASSPATH.

**Example: ApplicationContext context = new ClassPathXmlApplicationContext ("Beans.xml");**

**3. WebXmlApplicationContext -** This container loads the XML file with definitions of all beans from within a web application.

**Example: XmlWebApplicationContext context = new XmlWebApplicationContext();**

**context.setConfigLocation("/WEB-INF/beans.xml");**

**context.setServletContext(request.getServletContext());**

**context.refresh();**

**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 (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Consider an example similar to the employee address example stated previously.

**public class TextEditor {**

**private SpellChecker spellChecker;**

**public TextEditor() {**

**spellChecker = new SpellChecker();**

**}**

**}**

As in the previous example dependency between TextEditor and SpellChecker is high. This is solved by the Dependency Injection.

**public class TextEditor {**

**private SpellChecker spellChecker;**

**public TextEditor(SpellChecker spellChecker) {**

**this.spellChecker = spellChecker;**

**}**

**}**

By inversion of control, the dependency between TextEditor and SpellChecker is minimised.

We have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a Class Constructor. Thus the flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

The second method of injecting dependency is through Setter Methods of the TextEditor class where we will create a SpellChecker instance. This instance will be used to call setter methods to initialize TextEditor's properties.

DI exists in two major variants:

#### **Constructor-based DI**

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

If you are passing a value directly then you should use value attribute.

**<beans>**

**<bean id = "exampleBean" class = "examples.ExampleBean">**

**<constructor-arg type = "int" value = "2001"/>**

**<constructor-arg type = "java.lang.String" value = "Zara"/>**

**</bean>**

**</beans>**

Another way to pass values to constructor is:

**<beans>**

**<bean id = "exampleBean" class = "examples.ExampleBean">**

**<constructor-arg index = "0" value = "2001"/>**

**<constructor-arg index = "1" value = "Zara"/>**

**</bean>**

**</beans>**

In case you are passing a reference to an object, you need to use ref attribute of <constructor-arg> tag

**<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">**

**<constructor-arg ref = "spellChecker"/>**

**</bean>**

#### **Setter-based DI**

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

**<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">**

**<property name = "spellChecker" ref = "spellChecker"/>**

**</bean>**

**<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>**

**EXERCISE:**

1. Implement dependency injection and IoC in a basic Spring application and explore the bean lifecycle ( Create a Maven project ).
2. Implement register and login flow using spring MVC.

### **CODE LINK:**

<https://github.com/M-Abishaik/Zterns/tree/master/Spring%20MVC>