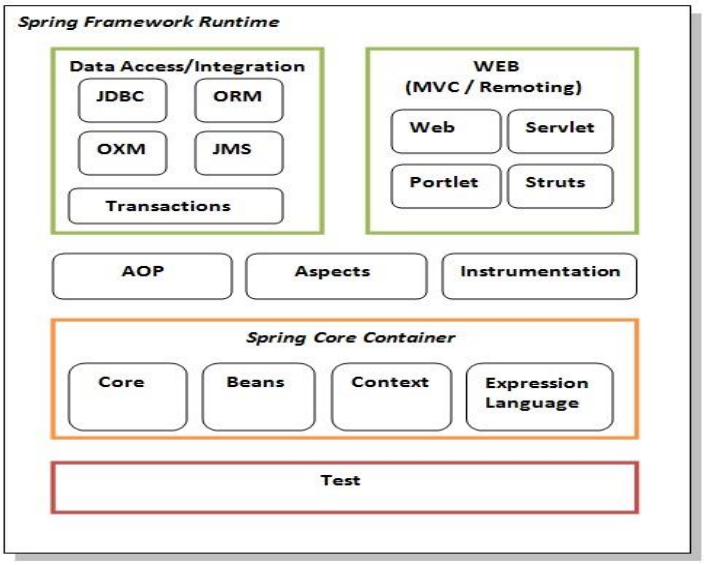
Spring	1. **Overview** 2. **Architecture** 3. **IoC Containers** 4. **Bean Definition** 5. **Bean Scopes** 6. **Bean Life Cycle** 7. **Bean Post Processors** 8. **Bean Definition Inheritance** 9. **Dependency Injection** 10. **Injecting Inner Beans** 11. **Injecting Collection** 12. **Beans Auto-Wiring** 13. **Annotation Based Configuration** 14. **Java Based Configuration** 15. **Event Handling in Spring** 16. **Custom Events in Spring** 17. **AOP with Spring Framework** 18. **JDBC Framework** 19. **Transaction Management** 20. **Web MVC Framework**

1. Overview

Why spring	**Benefits/Applications**
* Spring framework is an open source and most popular Java platform that provides **comprehensive infrastructure support** for developing robust Java applications very easily and very rapidly. * It is used to create high performing Applications, easily testable, and reusable code. *It is an open source Java platform. *It is lightweight in size(2MB basic version)	**POJO Based -** We can develop enterprise-class applications using POJOs. And it needs only robust servlet container such as Tomcat or some commercial product (No EJB container required for bean). **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 -** like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies. **Testablity -** It becomes easier to use dependency injection for injecting test data. **Web MVC -** Spring's web framework is a well-designed web MVC framework.(better than struts based). **Central Exception Handling -** Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO) **Lightweight -** 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). **Dependency Injection (DI)** DI is a special feature in spring which create loosely coupled application. Dependency Injection is merely one concrete example of Inversion of Control.

2. Architecture (Spring module)



Core Container	Data Access/Integration	Web	Miscellaneous
The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows − 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.	**The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −** 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 mapping APIs, including JPA, JDO, Hibernate, and iBatis. 3.The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX 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.	**The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −** 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 applications. 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.	**There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −** *1.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. 2.The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework. 3.The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers. 4.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. 5.The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

3. IoC Containers

The Spring container is 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 to manage the components that make up an application. These objects are called Spring Beans. 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.

Hello.java (POJO class)	Beans.xml (Creation of bean related to POJO classes)

public class **HelloWorld** {
private String message;
 	
public void setMessage(String message){
this.message = message;
}
public void getMessage(){
System.out.println("Your Message : " + message);
}
	<?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.xx.HelloWorld**">
	<property name = "message" value = "Hello World!"/>
	</bean>
	
	</beans>

Container	Initialization of Bean object by containers

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.
	public class MainApp {
	public static void main(String[] args) {
	
	XmlBeanFactory factory =
	new XmlBeanFactory (new assPathResource("Beans.xml"));
	
	HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");
	
	obj.getMessage(); //Hello World
	}
	}

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

4. Bean Definition

The objects which 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.

configuration metadata	
Bean definition contains the information called **configuration metadata**, which is needed for the container to know the following • How to create a bean • Bean's lifecycle details • Bean's dependencies	Following are the three important methods to provide configuration metadata to bean for the Spring Container − • XML based configuration file. • Annotation-based configuration • Java-based configuration

Beans.xml (configuration metadata)

```
<?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"
default-init-method = "init"            →these lifecycle hook method is applicable to every bean bydefault
default-destroy-method = "destroy">     →these lifecycle hook method is applicable to every bean bydefault

<bean
        id = "helloWorld"                       → bean unique id
        class = "com.xxx.HelloWorld"            → absolute path of a POJO class
        scope = "singleton/prototype
                /request/session/global-session" → what kind of object need to create
        init-method = "init"            → bean lifecycle hook onInit, it will call init() method of HelloWorld.class
        destroy-method = "destroy">     → bean lifecycle hook

   <property name = "message" ref = "bar"/>   → property value by referance of other bean object (setter DI)
   <property name = "message" value = "Hello World!"/>     → setting property directoly
   <constructor-arg ref = "bar"/>                          → constructor with reference (constructor DI)
  <constructor-arg index = "0" type = "int" value = "2001"/> → direct value/with respect to arg index

  </bean>

  <bean id = "bar" class = "x.y.Bar">
            <property name = "name" value = "xyz"/>
  </bean>

  <bean id = "bar" class = "x.y.Bar"
            p:name='xyz'>                   →setting property using p: and constructor with c: name space
  </bean>

</beans>
```

5. **Bean Scopes and**
6. **Bean life cycle**

Bean Scopes: Force spring container to return new instance each time or return same object each time.

Bean life cycle :-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

scope	description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
Session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

<table>
<tr><th>singleton</th><th>prototype</th></tr>
<tr><td colspan="2">

```
public class HelloWorld {
        private String message;

        public void setMessage(String message){
        this.message  = message;
        }
        public void getMessage(){
        System.out.println("Your Message : " + message);
        }
        public void init(){
            System.out.println("Bean is going through init.");
        }
         public void destroy() {
            System.out.println("Bean will destroy now.");
        }
}
```

</td></tr>
<tr><td>

```
<?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.xx.HelloWorld"
scope='singleton' init-method = "init" destroy-method =
"destroy">
    <property name = "message" value = "Hello World!"/>
  </bean>

</beans>
```

</td><td>

```
<?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.xx.HelloWorld"
scope='prototype' init-method = "init" destroy-method =
"destroy">
    <property name = "message" value = "Hello World!"/>
  </bean>

</beans>
```

</td></tr>
<tr><td>

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

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");
 objA.setMessage("I'm object A");
 objA.getMessage();  // I'm object A

 HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
 objA.getMessage();  // I'm object A
  }
}
```

</td><td>

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

 HelloWorld objA = (HelloWorld) context.getBean("helloWorld");
 objA.setMessage("I'm object A");
 objA.getMessage();  // I'm object A

 HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
    objB.getMessage(); // null
  }
}
```

</td></tr>
</table>

7. Bean Post Processors

The **BeanPostProcessor** interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic, etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the order property provided the BeanPostProcessor implements the Ordered interface.

The BeanPostProcessors operate on bean (or object) instances, which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An ApplicationContext automatically detects any beans that are defined with the implementation of the BeanPostProcessor interface and registers these beans as postprocessors, to be then called appropriately by the container upon bean creation. (**Example**:-)

```
public class HelloWorld {
  private String message;

  public void setMessage(String message){
    this.message  = message;
  }
  public void getMessage(){
    System.out.println("test msg”);
  }
  public void init(){
    System.out.println("bean init method");
  }
  public void destroy(){
    System.out.println("Bean will destroy now.");
  }
}
```

```
//beans.xml

<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.xx.HelloWorld"
    init-method = "init" destroy-method = "destroy">
    <property name = "message" value = "Hello World!"/>
  </bean>

  <bean class = "com.xx.InitHelloWorld" />

</beans>
```

```
public class InitHelloWorld implements BeanPostProcessor {

  public Object postProcessBeforeInitialization(Object bean, String beanName)
    throws BeansException {
    System.out.println("BeforeInitialization : " + beanName);
    return bean;  // you can return any other object as well
  }

  public Object postProcessAfterInitialization(Object bean, String beanName)
    throws BeansException {
    System.out.println("AfterInitialization : " + beanName);
    return bean;  // you can return any other object as well
  }

}
```

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

    HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
    obj.getMessage();
    context.registerShutdownHook();

   //** registerShutdownHook  :it will ensures a graceful shutdown and calls the relevant destroy methods.
  }
}
```

8. Bean Definition Inheritance

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. **parent** attribute is used for Bean based inheritance. **Bean definition template**, which can be used by other child bean definitions without putting much effort. Instead of using class just use **abstract = "true".**

Bean definition Inheritance (Example)	**Bean definition template (Example)**

public class **HelloWorld** {
private String message1;
private String message2;
 	
//setter and getter
}
	public class **HelloIndia** {
	private String message1;
	private String message2;
	private String message3;
	
	//setter and getter
	}

<?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.xx.HelloWorld" <property name = "message1" value = "Hello World!"/> <property name = "message2" value = "Hello Second World!"/> <property name = "message3" value = "Namaste India!"/> </bean> <bean id = "**helloIndia**" class = "com.xx.HelloIndia" **parent = "beanTeamplate"**> <property name = "message1" value = "Hello India!"/> <property name = "message3" value = "Namaste India!"/> </bean> </beans>	<?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 = "**beanTeamplate**" **abstract = "true">** <property name = "message1" value = "Hello World!"/> <property name = "message2" value = "Hello Second World!"/> <property name = "message3" value = "Namaste India!"/> </bean> <bean id = "**helloIndia**" class = "com.xx.HelloIndia" **parent = "beanTeamplate"**> <property name = "message1" value = "Hello India!"/> <property name = "message3" value = "Namaste India!"/> </bean> </beans>

public class MainApp {
public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
 	
HelloWorld objA = (HelloWorld) context.getBean("**helloWorld**");
objA.getMessage1(); // Hello World!
objA.getMessage2(); // Hello Second World!
 	
HelloIndia objB = (HelloIndia) context.getBean("**helloIndia**");
objB.getMessage1(); // Hello India!
objB.getMessage2(); // Hello second world
objB.getMessage3(); // Namaste India!
}
}
	public class MainApp {
	public static void main(String[] args) {
	ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
	
	HelloWorld objA = (HelloWorld) context.getBean("**helloIndia**");
	objA.getMessage1(); // Hello India!
	objA.getMessage2(); //Hello second world!
	}
	}

9. Dependency Injection and
10. Injecting inner beans

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.

Plane java based dependency (using new Keyword)	IOC based dependency

public class **TextEditor** {
 	
private **SpellChecker** spellChecker;
public **TextEditor**() {
spellChecker = **new** SpellChecker();
}
}
	public class **TextEditor** {
	private **SpellChecker** spellChecker;
	
	public **TextEditor**(**SpellChecker** spellChecker) {
	this.spellChecker = spellChecker;
	}
	}

*Here TexEditor has full control to initialize SpellChecker instance **.**	*Here SpellChecker instance will be injected through external source to class constructor (via IOC container)

Constructor dependency Injection

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker) {
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }
   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}



// ------------------------ SpellChecker.class -------------

public class SpellChecker {
   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }
}

// ------------------------ Beans.xml-------------

<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 = "textEditor" class = "com.xx.TextEditor">
      <constructor-arg ref = "spellChecker"/>
   </bean>

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

                    OR(Inner bean Injection)
<bean id = "textEditor" class = "com.xx.TextEditor">
    < constructor-arg  >
      <bean id = "spellChecker" class = "com.xx.SpellChecker"/>
     </ constructor-arg  >
    </bean>
</bean>

</beans>


// ------------------------ Main.class-------------

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

      TextEditor te = (TextEditor) context.getBean("textEditor");
      te.spellCheck();
   }
}
```

Setter dependency injection

```
public class TextEditor {
   private SpellChecker spellChecker;

   public void setSpellChecker(SpellChecker spellChecker) {
      this.spellChecker = spellChecker;
   }
   public SpellChecker getSpellChecker() {
      return spellChecker;
   }
   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}

// ------------------------ SpellChecker.class -------------


public class SpellChecker {
   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }
}


// ------------------------ Beans.xml-------------

<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 = "textEditor" class = "com.xx.TextEditor">
      <property name = "spellChecker" ref = "spellChecker"/>

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


                    OR(Inner bean injection)
<bean id = "textEditor" class = "com.xx.TextEditor">
    <property name = "spellChecker">
      <bean id = "spellChecker" class = "com.xx.SpellChecker"/>
     </property>
    </bean>
</bean>



</beans>

// ------------------------ Main.class-------------

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

      TextEditor te = (TextEditor) context.getBean("textEditor");
      te.spellCheck();
   }
}
```

11. Injecting Collection

if we want to pass multiples values like Java Collection types such as List, Set, Map, and Properties.

JavaCollection .class

```
public class JavaCollection {
  List addressList;
  Set  addressSet;
  Map  addressMap;
  Properties addressProp;

  // setter and getter

}
```

Beans.xml (setting collection properties of java class through bean)

```
<?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">

  <!-- Definition for javaCollection -->
  <bean id = "javaCollection" class = "com.xx.JavaCollection">

    <!-- results in a setAddressList(java.util.List) call -->
    <property name = "addressList">
      <list>
        <value>INDIA</value>
        <value>Pakistan</value>
      </list>
    </property>

    <!-- results in a setAddressSet(java.util.Set) call -->
    <property name = "addressSet">
      <set>
        <value>INDIA</value>
        <value>Pakistan</value>
      </set>
    </property>

    <!-- results in a setAddressMap(java.util.Map) call -->
    <property name = "addressMap">
      <map>
        <entry key = "1" value = "INDIA"/>
        <entry key = "2" value = "Pakistan"/>
      </map>
    </property>

    <!-- results in a setAddressProp(java.util.Properties) call -->
    <property name = "addressProp">
      <props>
        <prop key = "one">INDIA</prop>
        <prop key = "two">Pakistan</prop>
      </props>
    </property>
  </bean>

</beans>
```

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

    jc.getAddressList();
    jc.getAddressSet();
    jc.getAddressMap();
    jc.getAddressProp();
  }
}
```

12. Beans Auto-Wiring

We 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. if there will be more than one properties with other dependency then autowire will take care of all automatically.

Auto-wiring mode	Description
no	This is default setting which means no autowiringThis is default setting which means no autowiring.
byName	Autowiring by property name. Spring container looks at the properties of the beans on which autowire 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.
byType	Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file
constructor	Similar to byType, but type applies to constructor arguments
autoselect	Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

Example:1- [autowire with byName,bytype]

```
public class TextEditor {
// depandancy Injection by property method
  private SpellChecker spellChecker;
  private String name;

  public void setSpellChecker( SpellChecker spellChecker ){
    this.spellChecker = spellChecker;
  }
  public SpellChecker getSpellChecker() {
    return spellChecker;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void spellCheck() {
    spellChecker.checkSpelling();
  }
}
```

```
public class SpellChecker {
  public void checkSpelling() {
    System.out.println("Inside checkSpelling." );
  }
}
```

beans.xml (normal DI via ref)

```
<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 = "textEditor" class = "com.xx.TextEditor">
    <property name = "name" value = "Generic Text Editor" />
    <property name = "spellChecker" ref = "spellChecker" />
  </bean>

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

</beans>
```

beans.xml (autowiring via byname → property name and other bean name should be same)

```
<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 = "textEditor" class = "com.xx.TextEditor"
    autowire = "byName">
    <property name = "name" value = "Generic Text Editor" />
  </bean>
  <bean id = "spellChecker" class = "com.xx.SpellChecker">
</bean>

</beans>
```

beans.xml (normal DI via ref)

```
<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 = "textEditor" class = "com.xx.TextEditor">
    <property name = "name" value = "Generic Text Editor" />
    <property name = "spellChecker" ref = "spellChecker" />
  </bean>

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

</beans>
```

beans.xml (autowiring via byType → property Type and other bean type should be same)

```
<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 = "textEditor" class = "com.xx.TextEditor"
    autowire = "byType">
    <property name = "name" value = "Generic Text Editor" />
  </bean>
  <bean id = "spellChecker" class = "com.xx.SpellChecker">
</bean>

</beans>
```

```
public class MainApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
    TextEditor te = (TextEditor) context.getBean("textEditor");
    te.spellCheck();
  }
}
```

Example 2: [autowire with constructor,autodetect]

```
public class TextEditor {
// depandancy Injection by constructor method

  private SpellChecker spellChecker;
  private String name;

  public TextEditor( SpellChecker spellChecker, String name ) {
    this.spellChecker = spellChecker;
    this.name = name;
  }
  public SpellChecker getSpellChecker() {
    return spellChecker;
  }
  public String getName() {
    return name;
  }
  public void spellCheck() {
    spellChecker.checkSpelling();
  }
}
```

```
public class SpellChecker {
  public void checkSpelling() {
    System.out.println("Inside checkSpelling." );
  }
}
```

beans.xml (normal DI via ref)

```
<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 = "textEditor" class = "com.xx.TextEditor">
    <constructor-arg  value = "Generic Text Editor"/>
    <constructor-arg  ref = "spellChecker" />
  </bean>

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

</beans>
```

beans.xml (autowiring via **constructor → constructor Type and other bean type should be same)**

```
<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 = "textEditor" class = "com.xx.TextEditor"
    autowire = "constructor">
    <property name = "name" value = "Generic Text Editor" />
  </bean>
  <bean id = "spellChecker" class = “com.xx.SpellChecker">
</bean>

</beans>
```

beans.xml (normal DI via ref)

```
<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 = "textEditor" class = "com.xx.TextEditor">
    <constructor-arg  value = "Generic Text Editor"/>
    <constructor-arg  ref = "spellChecker" />
  </bean>

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

</beans>
```

beans.xml (autowiring via **autodetect →spring tries to wire first with constructor type if fails then try with byType)**

```
<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 = "textEditor" class = "com.xx.TextEditor"
    autowire = "autodetect">
    <property name = "name" value = "Generic Text Editor" />
  </bean>
  <bean id = "spellChecker" class = “com.xx.SpellChecker">
</bean>

</beans>
```

```
public class MainApp {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
    TextEditor te = (TextEditor) context.getBean("textEditor");
    te.spellCheck();
  }
}
```

13. Annotation Based Configuration

Starting from Spring 2.5 it became possible to configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

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.

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.

```
<beans xmlns = "=====xsd=====">

  <context:annotation-config/>
  <!-- bean definitions go here -->

</beans>
```

Anotations	Descriptions
@Required	The **@Required** annotation applies to bean property setter methods. There was dependency check feature before spring 3.0 (xml based that was deprecated ,we can achieve this feature by using @Required)
@Autowired(required=true)	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

Example:[@Required,@Autowired,@Qualifier,@Resource,@PostConstruct,@preDestroy]

```
public class TextEditor {

  @Required
  private String name;

  @Autowired
  private SpellChecker spellChecker;

  @Autowired
  @Qualifier("wordChecker2")
  private WordChecker wordChecker;

  @Resource(name="wordChecker1")
  private WordChecker wordChecker2;

  public void spellCheck() {
    spellChecker.checkSpelling();
  }
@PostConstruct
  public void init(){
    System.out.println("Bean is going through init.");
  }
  @PreDestroy
  public void destroy(){
    System.out.println("Bean will destroy now.");
  }
}
-----------------------------------------------------------------
public class SpellChecker {
  public void checkSpelling(){     System.out.println("Inside
checkSpelling." );
  }
}
-----------------------------------------------------------------
public class WordChecker {
  public void checkWord(){
    S.O.P("Inside wordchecker." );
  }
}
```

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

  <context:annotation-config/>

  <bean id = "textEditor" class = "com.xx.TextEditor">
    <property name = "name" value = "Generic Text Editor" />  // required

  </bean>

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

  <bean id = "wordChecker1" class = "com.xx.WordChecker">
  </bean>

  <bean id = "wordChecker2" class = "com.xx.WordChecker">
  </bean>

</beans>
```

Note:-

@Autowire →(autowired typw byType)

@Autowire
@Qualifire("name") → autowire type byname

@Resource(name="name") →autowire type byName

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

    TextEditor textEditor = (Student) context.getBean("textEditor ");
    System.out.println(textEditor. spellCheck () );
    context.registerShutdownHook();   // *note→  registerShutdownHook will ensure a graceful shutdown and call the relevant destroy methods.
  }
}
```

14. Java Based Configuration

Java-based configuration option enables you to write most of your Spring configuration without XML (by using Java-based annotations).
* **AnnotationConfigApplicationContext** class is required for IOC container initialization to read Bean definitions file.

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

* **@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.

ConfigA.java

```
public class ConfigA {
  @Bean
   public ConfigChecker configChecker (){
     return new ConfigChecker ( );
   }
}
```

ConfigChecker.java

```
--------------------------
public class ConfigChecker {
   public void configChecker (){
     System.out.println("Inside configChecker." );
   }
}
```

TextEditorConfig.java

```
@Configuration                  → initialize bean configuration
@Import(ConfigA.class) → import other configuration class
public class TextEditorConfig {

   → Bean initialization with lifecycle hook and  constructor DI
   @Bean (initMethod = "init", destroyMethod = "cleanup" )
   public TextEditor textEditor(){
     return new TextEditor( spellChecker() );
   }

  @Bean   → bean initialization for SpellCheck
   public SpellChecker spellChecker(){
     return new SpellChecker( );
   }

   @Bean
   @Scope("prototype") → bean Scope setting
   public SpellChecker spellChecker(){
     return new SpellChecker( );
   }
}
```

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker){
     this.spellChecker = spellChecker;
   }
   public void spellCheck(){
     spellChecker.checkSpelling();
   }
public void init() {
     // initialization logic
   }
   public void cleanup() {
     // destruction logic
   }
}
```

```
public class SpellChecker {
   public void checkSpelling(){
     System.out.println("Inside checkSpelling." );
   }
}
```

```
public class MainApp {
   public static void main(String[] args) {
     ApplicationContext ctx =  new AnnotationConfigApplicationContext(TextEditorConfig.class);

     TextEditor te = ctx.getBean(TextEditor.class);
    ConfigChecker cc = ctx.getBean(ConfigChecker.class); → Get imported configuration bean info

     te.spellCheck();
   }
}
```

15. Event Handling in Spring

We have seen in all the topics that the core of Spring is the ApplicationContext, which manages the complete life cycle of the beans. The ApplicationContext publishes certain types of events when loading the beans. For example, a ContextStartedEvent is published when the context is started and ContextStoppedEvent is published when the context is stopped.

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. Hence, if a bean implements the ApplicationListener, then every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified.

Spring's event handling is single-threaded so if an event is published, until and unless all the receivers get the message, the processes are blocked and the flow will not continue. Hence, care should be taken when designing your application if the event handling is to be used.

Events	Description
ContextRefreshedEvent	This event is published when the ApplicationContext is either initialized or refreshed. This can also be raised using the refresh() method on the ConfigurableApplicationContext interface.
ContextStartedEvent	This event is published when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. You can poll your database or you can restart any stopped application after receiving this event.
ContextStoppedEvent	This event is published when the ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. You can do required housekeep work after receiving this event.
ContextClosedEvent	This event is published when the ApplicationContext is closed using the close() method on the ConfigurableApplicationContext interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
RequestHandledEvent	This is a web-specific event telling all beans that an HTTP request has been serviced.

Listening to Context Events

To listen to a context event, a bean should implement the ApplicationListener interface which has just one method onApplicationEvent(). So let us write an example to see how the events propagates and how you can put your code to do required task based on certain events.

Example: (event handling in spring)

```
public class HelloWorld {
  private String message;

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

```
public class CStartEventHandler
  implements ApplicationListener<ContextStartedEvent>{

  public void onApplicationEvent(ContextStartedEvent event) {
    System.out.println("ContextStartedEvent Received");
  }
}
```

```
public class CStopEventHandler
  implements ApplicationListener<ContextStoppedEvent>{

  public void onApplicationEvent(ContextStoppedEvent event) {
    System.out.println("ContextStoppedEvent Received");
  }
}
```

```
<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.xx.HelloWorld">
    <property name = "message" value = "Hello World!"/>
  </bean>

  <bean id = "cStartEventHandler"    class = “com.xx.CStartEventHandler"/>

  <bean id = "cStopEventHandler"    class = "com.xx.CStopEventHandler"/>

</beans>
```

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

    // Let us raise a start event.
    context.start();

    HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
    obj.getMessage();

    // Let us raise a stop event.
    context.stop();
  }
}
```

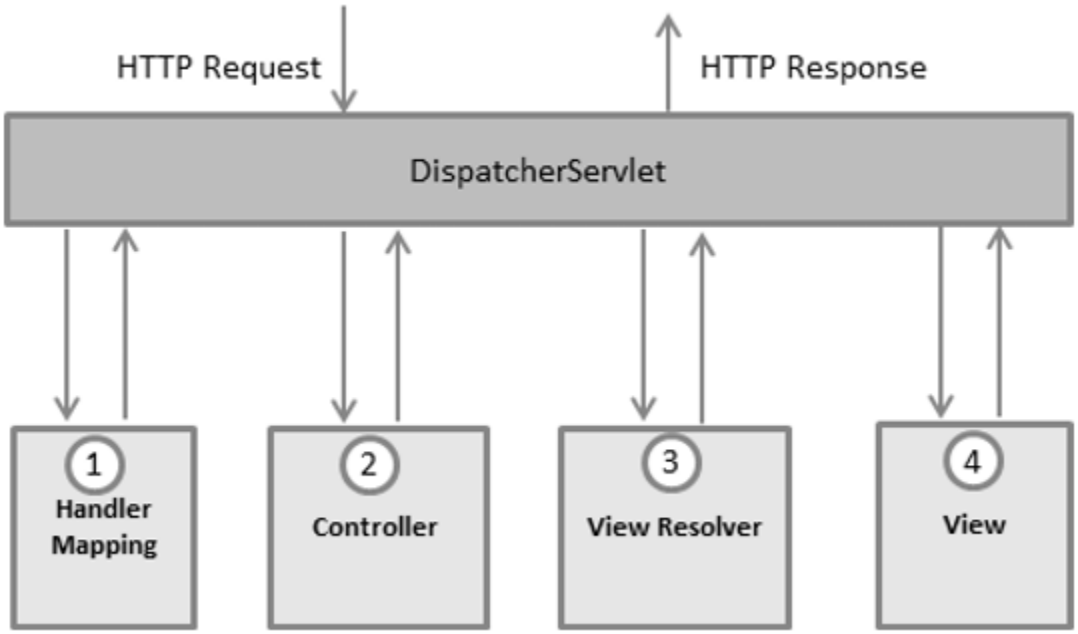
16. **Custom Events in Spring (read yourself)**
17. **AOP with Spring Framework (read yourself)**
18. **JDBC Framework (read yourself)**
19. **Transaction Management (read yourself)**

21. Web MVC Framework

The Spring Web MVC framework provides **Model-View-Controller (MVC) architectur**e 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.

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 –



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** which is an extension of the **plainApplicationContext** with some extra features necessary for web applications.

Required Configuration

You need to map requests that you want the **DispatcherServlet** to handle, by using a URL mapping in the **web.xml** file.

You can have as many **DispatcherServlets** as you want. Basically what you need to do is duplicate the configuration and give the servlet a different name (else it will overwrite the previous one), and have some separate configuration classes (or xml files) for it.

Your controllers shouldn't care in which **DispatcherServlet** they run neither should you include code to detect that.However while you can have multiple servlets in general there isn't much need for multiple servlets and you can handle it with a single instance of the DispatcherServlet.

The **web.xml** file will be kept in the **WebContent/WEB-INF** directory of your web application. Upon initialization of **WebAppFirst DispatcherServlet**, the framework will try to load the application context from a file named [servlet-name]-servlet.xml located in the application's WebContent/WEB-INF directory. In this case, our file will be **WebAppFirst-servlet.xml.**

*Default Location of web.xml and servlet:-**WebContent/WEB-INF/Web.xml**

WebContent/WEB-INF/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>
//Request URL or Respornse ulr
<servlet>
<servlet-name>WebAppFirst</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
<servlet-name> WebAppFirst </servlet-name>
<url-pattern>*.jsp</url-pattern>
</servlet-mapping>

--------------  second DispacherServlet config-----
<servlet>
<servlet-name>WebAppSecond</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name> WebAppSecond </servlet-name>
<url-pattern>*.jsp</url-pattern>
</servlet-mapping>

// if servlet will be in other location - -
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
</web-app>
```

WebContent/WEB-INF/ WebAppFirst-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">

  // Request URL – localhost:8080/hello.jsp
<context:component-scan base-package = "com.xx" />

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

Controller inside base-package WebAppFirst.java

```
-------------------------------------------

@Controller
@RequestMapping("/hello")
public class WebAppFirst {

  @RequestMapping(method = RequestMethod.GET)
  public String printHello(ModelMap model) {
    model.addAttribute("message", "Testing");

    return "hello";
  }
}
```

/WEB-INF/hello/hello.jsp

```
------------

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