Contents

[**https://www.digitalocean.com/community/tutorials?q=spring** 3](#_Toc169343333)

[Benefits of Using the Spring Framework 3](#_Toc169343334)

[Core Container 4](#_Toc169343335)

[Data Access/Integration 5](#_Toc169343336)

[Web 5](#_Toc169343337)

[Miscellaneous 5](#_Toc169343338)

[Install Spring suite 6](#_Toc169343339)

[Advantages of Spring Framework 6](#_Toc169343340)

[Inversion Of Control (IOC) and Dependency Injection 7](#_Toc169343341)

[Advantage of Dependency Injection 8](#_Toc169343342)

[Spring IoC Container 8](#_Toc169343343)

[Using BeanFactory 9](#_Toc169343344)

[Using ApplicationContext 10](#_Toc169343345)

[ApplicationContext Features 10](#_Toc169343346)

[Difference between BeanFactory and the ApplicationContext 11](#_Toc169343347)

[BEANS 12](#_Toc169343348)

[Bean scopes 12](#_Toc169343349)

[The singleton scope 13](#_Toc169343350)

[The prototype scope 13](#_Toc169343351)

[CustomBeanScope 14](#_Toc169343352)

[Bean Life Cycle 14](#_Toc169343353)

[By XML: 14](#_Toc169343354)

[By Programmatic Approach: 15](#_Toc169343355)

[By Annotation 15](#_Toc169343356)

[Dependency Injection in Spring 15](#_Toc169343357)

[Two ways to perform Dependency Injection in Spring framework 16](#_Toc169343358)

[Dependency Injection by Constructor Example 16](#_Toc169343359)

[Constructor Injection with Collection Example 16](#_Toc169343360)

[Inheriting Bean in Spring 17](#_Toc169343361)

[Dependency Injection by setter method 17](#_Toc169343362)

[Difference between constructor and setter injection 17](#_Toc169343363)

[Different ways to create bean 18](#_Toc169343364)

[Autowiring in Spring 18](#_Toc169343365)

[Modes of Autowiring 19](#_Toc169343366)

[byName autowiring mode 20](#_Toc169343367)

[byType autowiring mode 20](#_Toc169343368)

[constructor autowiring mode 20](#_Toc169343369)

[no autowiring mode 21](#_Toc169343370)

[autodetect autowiring mode 21](#_Toc169343371)

[Dependency Injection with Factory Method in Spring 21](#_Toc169343372)

[Annotation Based Configuration 21](#_Toc169343373)

[@Required Annotation 22](#_Toc169343374)

[@Autowired 22](#_Toc169343375)

[@Autowired on Setter Methods 22](#_Toc169343376)

[@Autowired on Properties 22](#_Toc169343377)

[@Autowired on Constructors 22](#_Toc169343378)

[@Autowired with (required = false) option 22](#_Toc169343379)

[@Qualifier 22](#_Toc169343380)

[JSR-250 Annotations 22](#_Toc169343381)

[@PostConstruct and @PreDestroy Annotations 23](#_Toc169343382)

[@Resource 23](#_Toc169343383)

[Spring - Java Based Configuration 23](#_Toc169343384)

[@Configuration & @Bean Annotations 23](#_Toc169343385)

[@Import 24](#_Toc169343386)

[Lifecycle Callbacks 25](#_Toc169343387)

[Specifying Bean Scope 25](#_Toc169343388)

[JdbcTemplate 26](#_Toc169343389)

[**Problems of JDBC API** 26](#_Toc169343390)

[**Advantage of Spring JdbcTemplate** 26](#_Toc169343391)

[**Spring Jdbc Approaches** 26](#_Toc169343392)

[**JdbcTemplate Class** 27](#_Toc169343393)

[ResultSetExtractor 27](#_Toc169343394)

[ResultSetExtractor Interface 28](#_Toc169343395)

[RowMapper 28](#_Toc169343396)

[RowMapper Interface 28](#_Toc169343397)

[Spring NamedParameterJdbcTemplate 28](#_Toc169343398)

[Simple example of named parameter query 28](#_Toc169343399)

[Spring - Transaction Management 28](#_Toc169343400)

[Spring Security 29](#_Toc169343401)

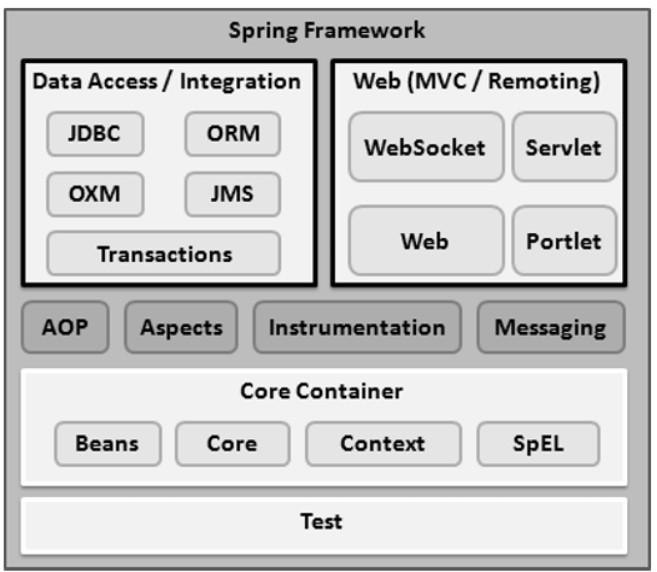
[**Spring Security Features** 29](#_Toc169343402)

## **https://www.digitalocean.com/community/tutorials?q=spring**

# 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).



# 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.

# 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.

# 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.

# 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.

Install Spring suite  
<https://www.geeksforgeeks.org/how-to-download-and-install-spring-tool-suite-spring-tools-4-for-eclipse-ide/?ref=lbp>

# Advantages of Spring Framework

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

##### 1) Predefined Templates

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

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

##### 2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

##### 3) Easy to test

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

##### 4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

##### 5) Fast Development

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

##### 6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as [JMS](https://www.javatpoint.com/jms-tutorial), [JDBC](https://www.javatpoint.com/java-jdbc), JPA and JTA.

##### 7) Declarative support

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

# Inversion Of Control (IOC) and Dependency Injection

<https://www.geeksforgeeks.org/spring-understanding-inversion-of-control-with-example/?ref=lbp>

Spring IoC (Inversion of Control) Container is the core of [Spring Framework](https://www.geeksforgeeks.org/introduction-to-spring-framework/). It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion Of Control.

Dependency Injection: Dependency Injection is a design pattern that allows the spring container to ‘inject’ objects into other objects or dependencies. In simple words, the control of creating objects and managing the spring components is taken care of by the Spring containers.

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain. Let's understand this with the following code:

**class** Employee{

Address address;

Employee(){

address=**new** Address();

}

}

In such case, there is dependency between the Employee and Address (tight coupling). In the Inversion of Control scenario, we do this something like this:

**class** Employee{

Address address;

Employee(Address address){

**this**.address=address;

}

}

Thus, IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

## Advantage of Dependency Injection

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

# Spring IoC Container

Spring IoC is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, objects dependencies are injected by other assembler objects. Spring IoC container is the program that **injects** dependencies into an object and make it ready for our use

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

That means if you want to use an IoC container in spring whether we need to use a BeanFactory or ApplicationContext. The BeanFactory is the most basic version of IoC containers, and the ApplicationContext extends the features of BeanFactory. The followings are some of the main features of Spring IoC,

* Creating Object for us,
* Managing our objects,
* Helping our application to be configurable,
* Managing dependencies

Example to understand IOC

<https://www.geeksforgeeks.org/spring-understanding-inversion-of-control-with-example/?ref=lbp>

# Using BeanFactory

The first and foremost thing when we talk about [**Spring**](https://www.geeksforgeeks.org/introduction-to-spring-framework/) is dependency injection which is possible because Spring is a container and behaves as a factory of **Beans**. Just like the **BeanFactory** interface is the simplest container providing an advanced configuration mechanism to instantiate, configure, and manage the life cycle of beans.

**Beans** are Java objects that are configured at run-time by [Spring IoC Container](https://www.geeksforgeeks.org/spring-ioc-container/). BeanFactory represents a basic**IoC container**

BeanFactory loads the bean definitions and dependency amongst the beans based on a configuration file (XML) or the beans can be directly returned when required using Java Configuration

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

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

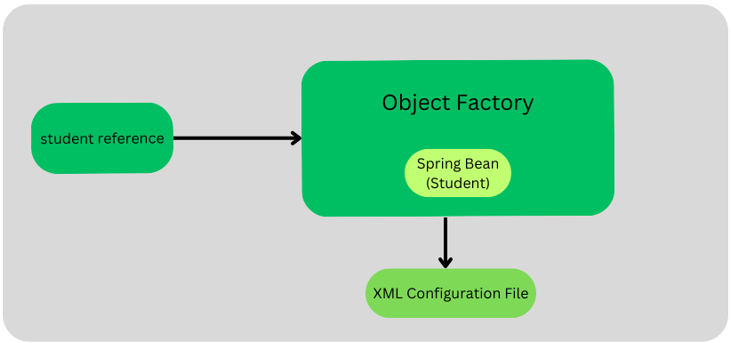
BeanFactory factory=new XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

XmlBeanFactory is depreciated use below code for creating beanfactory

BeanFactory factory = **new** ClassPathXmlApplicationContext("bean-factory-demo.xml");

Let’s understand the above code with visuals:



The program flow is something like this:

* First of all, the Bean factory reads the XML configuration file and as per the specifications defined in it, it creates the bean of the student POJO.
* Then the student reference asks for the student object from the object factory.
* Then finally, the spring object factory hands over the spring bean (student) to its reference. Here, note that the bean returned by the object factory is of “Object” type, so we have to typecast it into our desired bean.

# Using ApplicationContext

**ApplicationContext** belongs to the **Spring**framework. **Spring IoC contain**er is responsible for instantiating, wiring, configuring, and managing the entire life cycle of beans or objects. **BeanFactory** and **ApplicationContext**represent the Spring IoC Containers. **ApplicationContext** is the sub-interface of BeanFactory. It is used when we are creating an enterprise-level application or web application. ApplicationContext is the superset of BeanFactory, whatever features provided by BeanFactory are also provided by ApplicationContext.

## ApplicationContext Features

ApplicationContext provides basic features in addition to enterprise-specific functionalities which are as follows:

* Publishing events to registered listeners by resolving property files.
* Methods for accessing application components.
* Supports Internationalization.
* Loading File resources in a generic fashion.

1. AnnotationConfigApplicationContext container
2. AnnotationConfigWebApplicationContext
3. XmlWebApplicationContext
4. FileSystemXmlApplicationContext
5. ClassPathXmlApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context =

    new ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

# Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.BeanFactory and the org.springframework.context.ApplicationContext interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

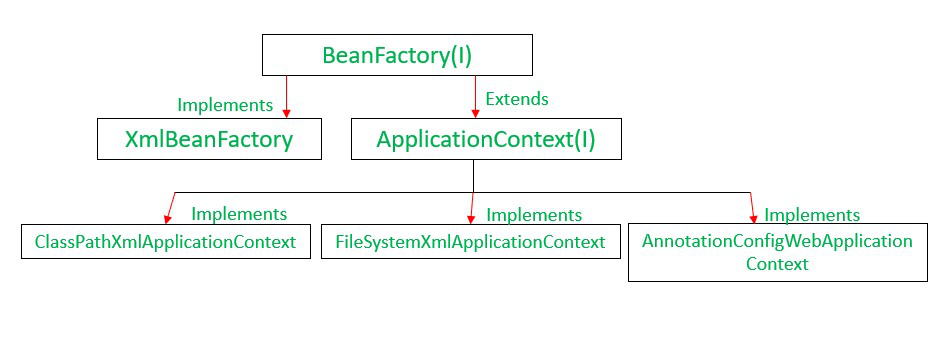
*BeanFactory***loads beans on-demand, while***ApplicationContext***loads all beans at startup**. Thus, *BeanFactory* is lightweight as compared to *ApplicationContext*

*ApplicationContext*enhances *BeanFactory* in a more framework-oriented style and provides several features that are suitable for enterprise applications.

For instance, it **provides**[messaging (i18n or internationalization)](https://www.baeldung.com/spring-classpathxmlapplicationcontext#2-internationalization-with-messagesource) functionality, [event publication](https://www.baeldung.com/spring-events) functionality, **annotation-based dependency injection**, and **easy integration with Spring AOP features**.

Apart from this, the *ApplicationContext* supports almost all types of bean scopes, but the *BeanFactory* only supports two scopes — *Singleton* and *Prototype*. Therefore, it's always preferable to use *ApplicationContext*when building complex enterprise applications

In short, the BeanFactory provides the configuration framework and basic functionality, while the ApplicationContext adds more enterprise-centric functionality to it.

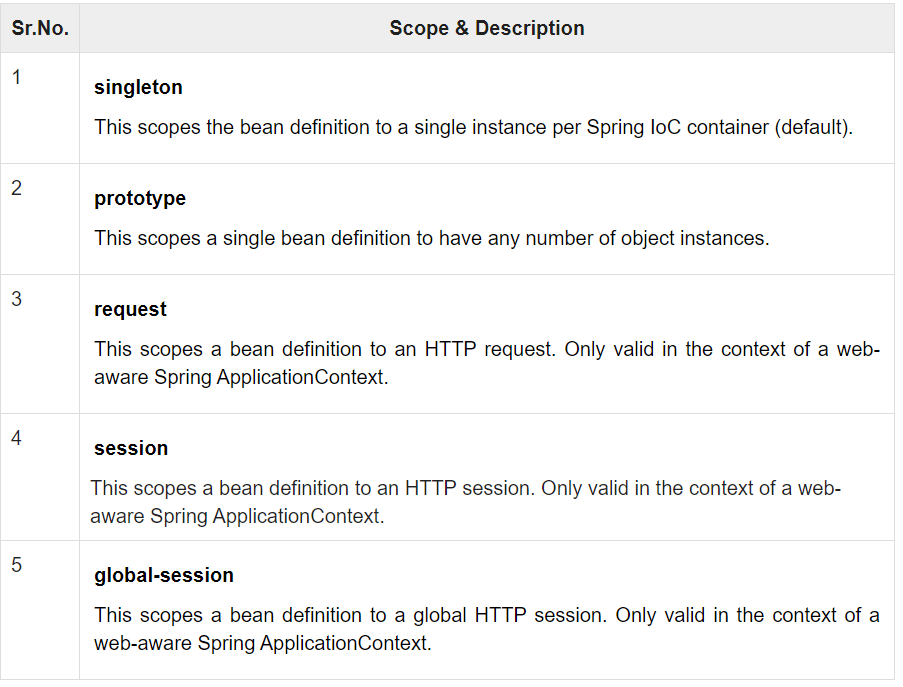


# BEANS

|  |  |
| --- | --- |
| 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. |

## Bean scopes

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



### The singleton scope

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton. However, when you need one and only one instance of a bean, you can set the scope property to singleton in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with singleton scope -->

<bean id = "..." class = "..." scope = "singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

### The prototype scope

If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the scope property to prototype in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with prototype scope -->

<bean id = "..." class = "..." scope = "prototype">

<!-- collaborators and configuration for this bean go here -->

</bean>

### CustomBeanScope

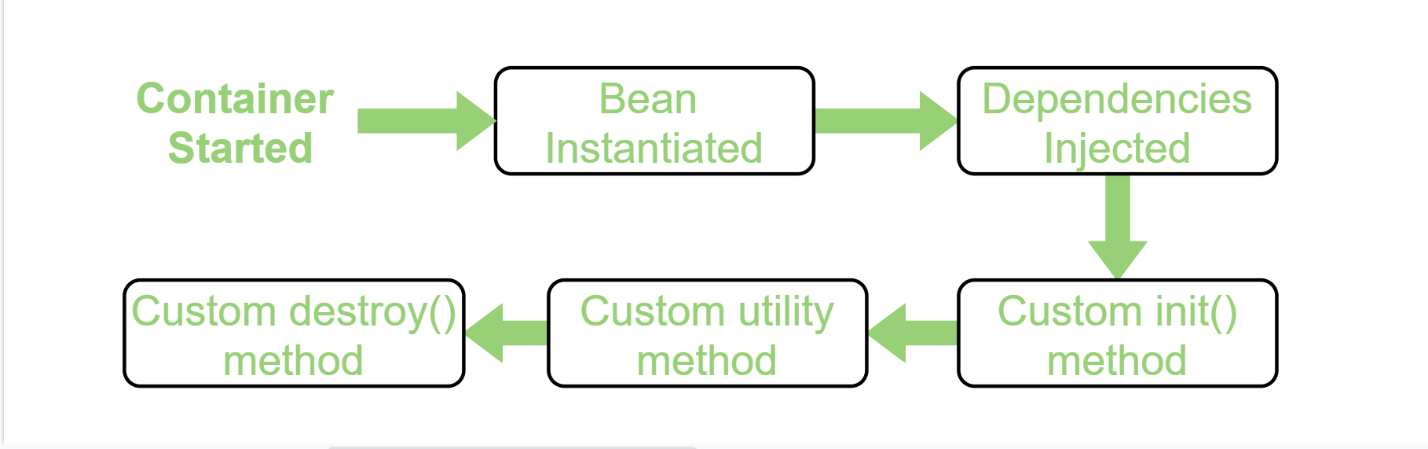
Let’s try to understand the requirement of custom bean scope with the help of an example. Suppose, if you are developing a multi-tenant system, you may want to provide a separate instance of a particular bean or set of beans for each tenant. Spring provides a mechanism for creating custom scopes for scenarios such as this.

Check below for example.

https://www.geeksforgeeks.org/custom-bean-scope-in-spring/?ref=lbp

## Bean Life Cycle

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.



### **By XML:**

##### Initialization callbacks

In the case of XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>

Following is the class definition −

public class ExampleBean {

public void init() {

// do some initialization work

}

}

##### Destruction callbacks

In the case of XML-based configuration metadata, you can use the destroy-method attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>

Following is the class definition −

public class ExampleBean {

public void destroy() {

// do some destruction work

}

}

### **By Programmatic Approach:**

Check below for example

<https://www.geeksforgeeks.org/bean-life-cycle-in-java-spring/?ref=lbp>

### By Annotation

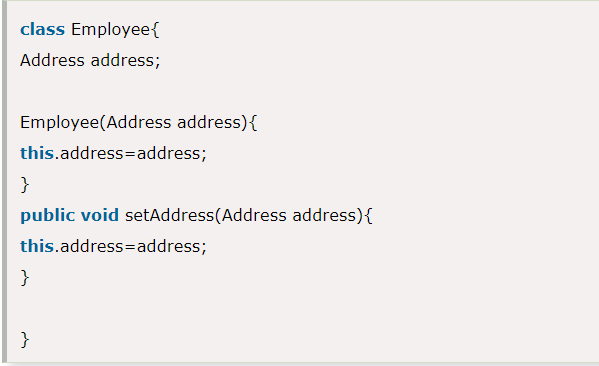
To provide the facility to the created bean to invoke custom **init()** method on the startup of a spring container and to invoke the custom **destroy()** method on closing the container, we need to annotate **init()** method by **@PostConstruct** annotation and **destroy()** method by **@PreDestroy** annotation.

To invoke the **destroy()** method we have to call the **close()** method of ConfigurableApplicationContext.

# Dependency Injection in Spring

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.

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, 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

## Dependency Injection by Constructor Example

We can inject the dependency by constructor. The <constructor-arg> subelement of <bean> is used for constructor injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

## Constructor Injection with Collection Example

We can inject collection values by constructor in spring framework. There can be used three elements inside the constructor-arg element.

It can be:

1. list
2. set
3. map

Each collection can have string based and non-string based values.

## Inheriting Bean in Spring

By using the parent attribute of bean, we can specify the inheritance relation between the beans. In such case, parent bean values will be inherited to the current bean.

# Dependency Injection by setter method

DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the bean-configuration file For this, the property to be set with the SDI is declared under the <property> tag in the bean-config file.

We can inject the dependency by setter method also. The **<property>** subelement of **<bean>** is used for setter injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

## Difference between constructor and setter injection

There are many key differences between constructor injection and setter injection.

1. Partial dependency: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. Overriding: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. Changes: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

**Check code for different examples**

# Different ways to create bean

create a Spring Bean in 3 different ways as follows:

1. Creating Bean Inside an XML Configuration File (beans.xml)
2. Using @Component Annotation
3. Using @Bean Annotation

**@Component**is an annotation that allows Spring to automatically detect the custom beans.

@Component is an annotation that allows Spring to detect our custom beans automatically.

In other words, without having to write any explicit code, Spring will:

* Scan our application for classes annotated with @Component
* Instantiate them and inject any specified dependencies into them
* Inject them wherever needed

**Checkcode for all the types of beancreation**

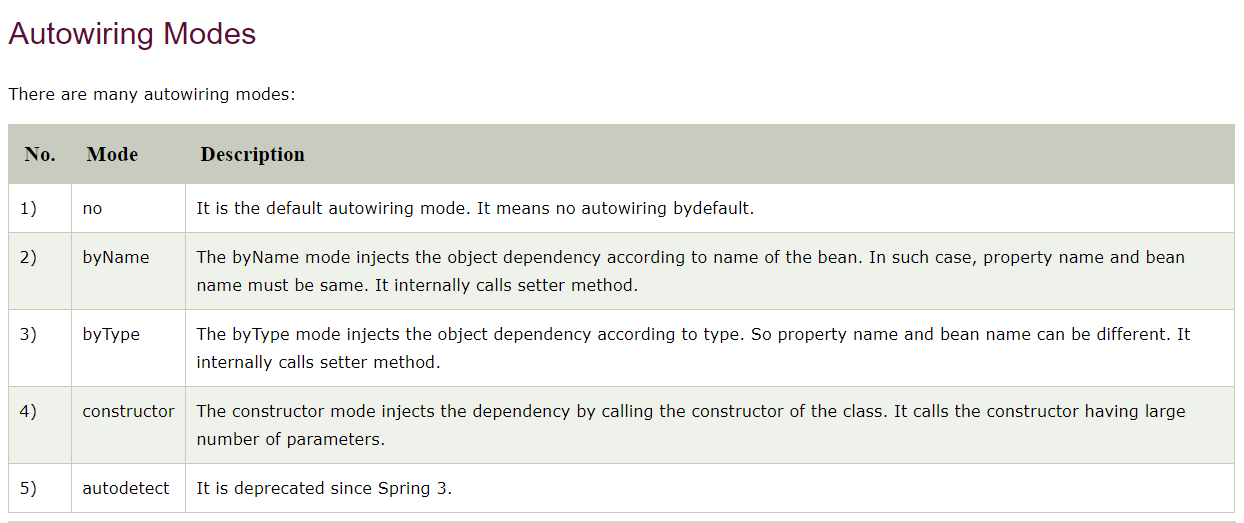
# Autowiring in Spring

**Autowiring** in the Spring framework can inject dependencies automatically. The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**. To enable Autowiring in the Spring application we should use[@Autowired](https://www.geeksforgeeks.org/spring-autowired-annotation/)annotation. Autowiring in Spring internally uses constructor injection. An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

Autowiring can't be used to inject primitive and string values. It works with reference only.

## **Modes of Autowiring**

| Modes | Description |
| --- | --- |
| No | This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring. |
| byName | It uses the name of the bean for injecting dependencies. |
| byType | It injects the dependency according to the type of bean. |
| Constructor | It injects the required dependencies by invoking the constructor. |
| Autodetect | The autodetect mode uses two other modes for autowiring – constructor and byType. |



<https://howtodoinjava.com/spring-core/spring-beans-autowiring-concepts/>

https://dzone.com/articles/autowiring-in-spring#:~:text=The%20XML%2Dconfiguration%2Dbased%20autowiring,The%20default%20mode%20is%20no.

### byName autowiring mode

In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are changing the name of the bean from b to b1.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

### byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.

It internally uses setter injection.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byType"></bean>

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

Let's see the code where are many bean of type B.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="b2" **class**="org.sssit.B"></bean>
3. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

In such case, it will throw exception.

### constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="constructor"></bean>

### no autowiring mode

In case of no autowiring mode, spring container doesn't inject the dependency by autowiring.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="no"></bean>

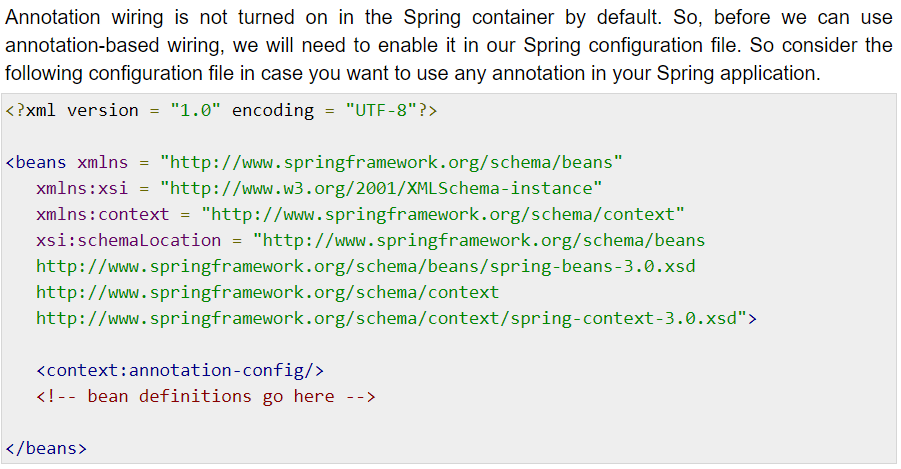
### autodetect autowiring mode

The autodetect mode uses two other modes for autowiring – constructor and byType. It first tries to autowire via the constructor mode and if it fails, it uses the byType mode for autowiring. It works in Spring 2.0 and 2.5 but is deprecated from Spring 3.0 onwards.

## Dependency Injection with Factory Method in Spring

Check programs

# Annotation Based Configuration



## @Required Annotation

<https://www.tutorialspoint.com/spring/spring_required_annotation.htm>

## @Autowired

<https://www.tutorialspoint.com/spring/spring_autowired_annotation.htm>

The **@Autowired** annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

## @Autowired on Setter Methods

You can use **@Autowired** annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform **byType** autowiring on the method.

## @Autowired on Properties

You can use **@Autowired** annotation on properties to get rid of the setter methods. When you will pass values of autowired properties using <property> Spring will automatically assign those properties with the passed values or references. So with the usage of @Autowired on properties

## @Autowired on Constructors

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file. Let us check the following example.

## @Autowired with (required = false) option

By default, the @Autowired annotation implies the dependency is required similar to @Required annotation, however, you can turn off the default behavior by using **(required=false)** option with @Autowired.

## @Qualifier

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

<https://www.tutorialspoint.com/spring/spring_qualifier_annotation.htm>

# JSR-250 Annotations

Spring also supports JSR-250 based annotations which include @PostConstruct, @PreDestroy and @Resource annotations. Though these annotations are not really required because you already have other alternates, yet let us get a brief idea about them.

## @PostConstruct and @PreDestroy Annotations

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

You can use **@PostConstruct** annotation as an alternate of initialization callback and **@PreDestroy** annotation as an alternate of destruction callback as explained in the below example.

## @Resource

The @Resource annotation in spring performs the autowiring functionality. This annotation follows the autowire=byName semantics in the XML based configuration i.e. it takes the name attribute for the injection.

This annotation takes an optional name argument. In case no name attribute is specified with this annotation, the default name is interpreted from the field-name or the setter method (i.e. the bean property name). Always **remember** that if the @Resource annotation doesn’t find the bean with the name it will automatically switch it’s autowiring technique to autowire=byType (i.e. @Autowired annotation).

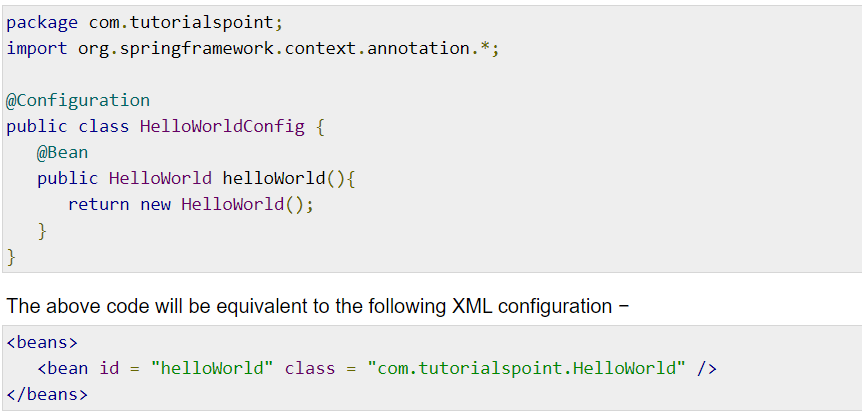
# Spring - Java Based Configuration

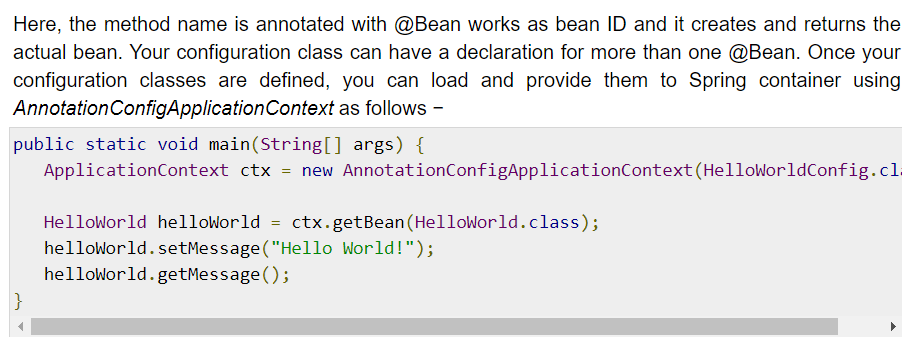
So far you have seen how we configure Spring beans using XML configuration file. If you are comfortable with XML configuration, then it is really not required to learn how to proceed with Java-based configuration as you are going to achieve the same result using either of the configurations available.

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

## @Configuration & @Bean Annotations

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context





## @Import

The **@Import** annotation allows for loading @Bean definitions from another configuration class. Consider a ConfigA class as follows −

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

You can import above Bean declaration in another Bean Declaration as follows −

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied as follows −

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

## Lifecycle Callbacks

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element −

public class Foo {

public void init() {

// initialization logic

}

public void cleanup() {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public Foo foo() {

return new Foo();

}

}

## Specifying Bean Scope

The default scope is singleton, but you can override this with the @Scope annotation as follows −

@Configuration

public class AppConfig {

@Bean

@Scope("prototype")

public Foo foo() {

return new Foo();

}

}

# JdbcTemplate

## **Problems of JDBC API**

The problems of JDBC API are as follows:

* We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
* We need to perform exception handling code on the database logic.
* We need to handle transaction.
* Repetition of all these codes from one to another database logic is a time consuming task.

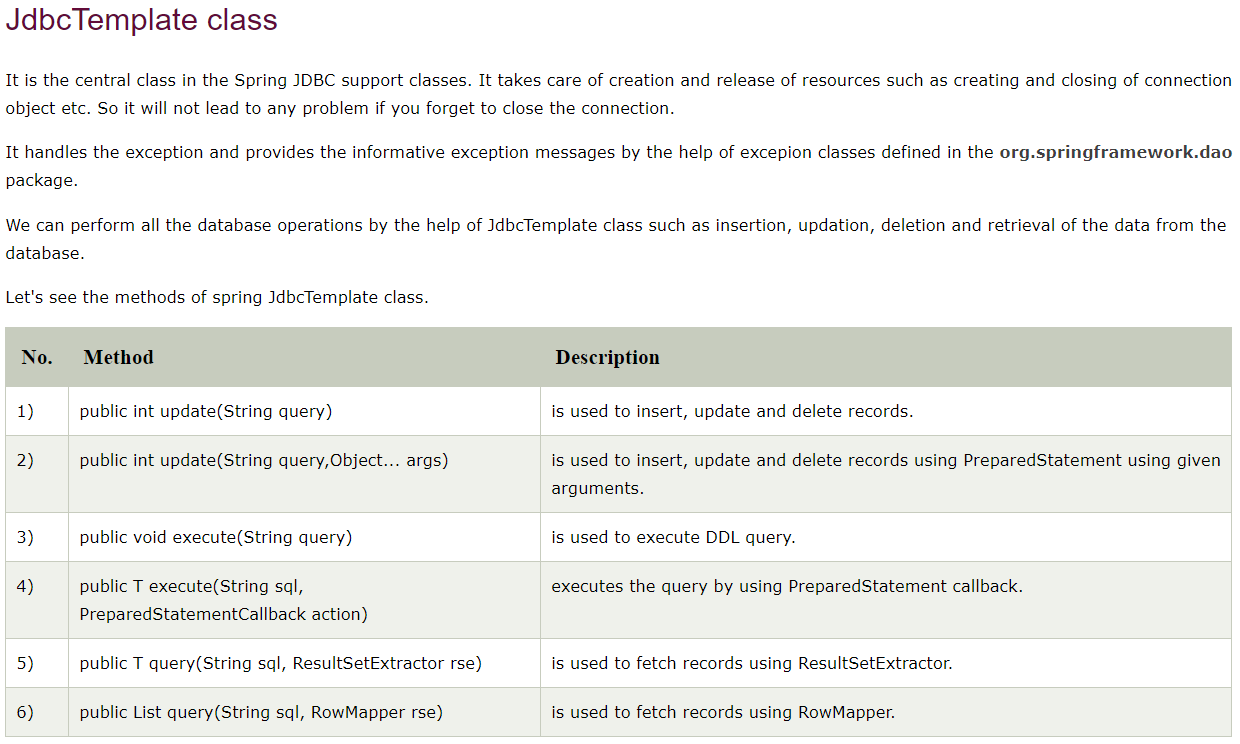
## **Advantage of Spring JdbcTemplate**

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

## **Spring Jdbc Approaches**

Spring framework provides following approaches for JDBC database access:

* JdbcTemplate
* NamedParameterJdbcTemplate
* SimpleJdbcTemplate
* SimpleJdbcInsert and SimpleJdbcCall



## **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.

# ResultSetExtractor

We can easily fetch the records from the database using **query()** method of **JdbcTemplate** class where we need to pass the instance of ResultSetExtractor.

#### Syntax of query method using ResultSetExtractor

1. **public** T query(String sql,ResultSetExtractor<T> rse)

## ResultSetExtractor Interface

**ResultSetExtractor** interface can be used to fetch records from the database. It accepts a ResultSet and returns the list.

# RowMapper

Like ResultSetExtractor, we can use RowMapper interface to fetch the records from the database using **query()** method of **JdbcTemplate** class. In the execute of we need to pass the instance of RowMapper now.

#### Syntax of query method using RowMapper

1. **public** T query(String sql,RowMapper<T> rm)

### RowMapper Interface

**RowMapper** interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

#### Advantage of RowMapper over ResultSetExtractor

RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.

# Spring NamedParameterJdbcTemplate

Spring provides another way to insert data by named parameter. In such way, we use names instead of ?(question mark). So it is better to remember the data for the column.

## Simple example of named parameter query

1. insert into employee values (:id,:name,:salary)

# Spring - Transaction Management

Check below url for transaction management

<https://www.tutorialspoint.com/spring/spring_transaction_management.htm>

# Spring Security

Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application.

This framework targets two major areas of application are authentication and authorization. Authentication is the process of knowing and identifying the user that wants to access.

**Authorization** is the process to allow authority to perform actions in the application.

We can apply authorization to authorize web request, methods and access to individual domain.

Technologies that support Spring Security Integration

Spring Security framework supports wide range of authentication models. These models either provided by third parties or framework itself. Spring Security supports integration with all of these technologies.

* HTTP BASIC authentication headers
* HTTP Digest authentication headers
* HTTP X.509 client certificate exchange
* LDAP (Lighweight Directory Access Protocol)
* Form-based authentication
* OpenID authentication
* Automatic remember-me authentication
* Kerberos
* JOSSO (Java Open Source Single Sign-On)
* AppFuse
* AndroMDA
* Mule ESB
* DWR(Direct Web Request)

# **Spring Security Features**

* LDAP (Lightweight Directory Access Protocol)
* Single sign-on
* JAAS (Java Authentication and Authorization Service) LoginModule
* Basic Access Authentication
* Digest Access Authentication
* Remember-me
* Web Form Authentication
* Authorization
* Software Localization
* HTTP Authorization

<https://www.javatpoint.com/spring-security-java-example>