**Introduction**

**Spring framework is an open source Java platform. It provides comprehensive infrastructure support for developing robust Java applications**. Spring framework promotes “Plain Old Java Objects” (POJO) based programming model. Spring provides extensions for building web application along with core features on top of the Java EE platform.

**Spring is a light-weight framework for the development of enterprise-ready applications. It provides following features:**

1. **Dependency Injection**
2. Declarative transaction management using Spring AOP.
3. Spring MVC web application and RESTful web service framework
4. Foundational support for JDBC, JPA, JMS
5. Mailing facilities.
6. Etc.

Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

* Spring 5
  + [Spring Modules](https://sites.google.com/site/corporatejavac1/Home/spring-modules)
  + [Getting Started](https://sites.google.com/site/corporatejavac1/Home/getting-started)
  + [Spring Core](https://sites.google.com/site/corporatejavac1/Home/spring-modules)
    - [IOC Container](https://sites.google.com/site/corporatejavac1/Home/ioc-container)
    - [Spring Beans](https://sites.google.com/site/corporatejavac1/Home/create-beans)
    - [Dependency0 Injection and IOC](https://sites.google.com/site/corporatejavac1/Home/dependency-injection-and-autowiring)
    - [Bean Autowiring](https://sites.google.com/site/corporatejavac1/Home/bean-autowiring)
    - [Auto Discovery](https://sites.google.com/site/corporatejavac1/Home/auto-discovery)
  + [Data Access and Integration](https://sites.google.com/site/corporatejavac1/Home/spring-modules)
    - [Spring DAO](https://sites.google.com/site/corporatejavac1/Home/spring-dao)
    - [Spring JDBC](https://sites.google.com/site/corporatejavac1/spring/spring-jdbc)
    - [Spring ORM](https://sites.google.com/site/corporatejavac1/spring/spring-orm)
  + [Spring AOP](https://sites.google.com/site/corporatejavac1/spring/spring-aop-1)
  + [Web](https://sites.google.com/site/corporatejavac1/Home/spring-modules)
    - [Spring Web MVC](http://www.google.com/url?q=http%3A%2F%2Fsmvc.sunilos.com%2F&sa=D&sntz=1&usg=AFQjCNF_yjGVUo5UvgcTLQeIrutHHwB90Q)

# Spring Modules

Spring framework consists of multiple modules. These modules are Core, Beans, Context, Expression Language (EL), AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc. These modules are grouped into Test, Core Container, AOP, Aspects, Instrumentation, Data Access / Integration, and Web.

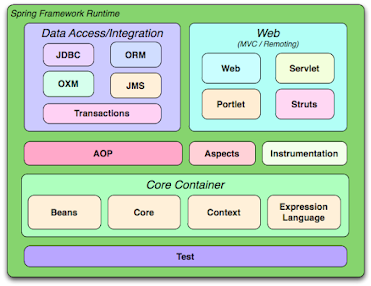
[](https://www.google.com/url?q=https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/images/spring-overview.png&sa=D&sntz=1&usg=AFQjCNEKjuRo86m7BCMykom8R4LeIYkWzQ)

Figure: Spring Modules

## Core Container

This layer provides IOC container that manages bean life cycle. It provides dependency injection mechanism. This layer will be used by all other spring modules and groups. It consists of the Core, Beans, Context, and Expression Language modules.

* **Core and Beans modules provides basic feature of bean life cycle management and dependency injection. It provides BeanFactory IOC container.**
* **Context module is built over Core and Beans. It inherits the features of bean module and provides application context container**. It supports internationalization (i18n), event propagation, EJB, JMX,
* **Expression Language** module provides expression language for querying and manipulating object graph from IOC container. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. It is used to get and set properties, method invocation, access collection, and logical/arithmetic operations.

## Data Access/Integration

This layer is used to integrate database ORM frameworks, object to XML binding frameworks, massaging services and handling database transaction s. The Data Access/Integration layer consists of JDBC, ORM, OXM, JMS and Transaction modules.

* **JDBC module** provides an abstraction layer of jdbc. Abstraction layer removes the dependency of core api call of jdbc and database vender specific exception handling.
* **ORM module** integrations spring with popular object-relational mapping tools. Popular ORM tools are JPA, JDO, Hibernate, and iBatis. It provides simple declarative transaction handling.
* **OXM module** provides abstraction layer to support Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
* **Java Messaging Service (JMS) module** provides features for producing and consuming messages.
* **Transaction module** supports programmatic and declarative transaction management.

## Web

This layer is used to integrate spring with web application and its framewoeks. Web layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

* **Web module** integrates spring with a web application. It initializes application context container for web application, It provides basic web application features like multipart file-upload.
* **Web-Servlet module** contains model-view-controller (MVC) implementation of spring for web applications. Spring MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.
* **Web-Struts** **module** integrates spring container with struts framework.
* **Web-Portlet module** provides the MVC implementation to be used in a portlet environment.

## AOP and Instrumentation

* **AOP module** provides aspect-oriented programming implementation where you can use advices, pointcuts etc. to decouple the code.
* **Aspects module** provides integration with AspectJ.
* **Instrumentation module** provides class instrumentation support and classloader implementations to be used in certain application servers.

## Test

Test module supports testing of spring components with JUnit or TestNG. It provides consistent loading of application context testing. It also provides mock objects that you can use to test your code in isolation.

# Getting Started

**Spring is providing IOC container. Container creates object of a simple or complex java class and manages object life cycle. Objects created and manages by spring IOC container is called bean.**

Lets configure and create instance of a simple [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) POJO class. POJO is stand for plain old java object. POJO contains private attributes and its setter and getter methods.

Lets follow given steps to create instance of User class by spring container.

Step 1.

Create Eclipse Maven java project

See url [http://www.raystec.com/Home/java-development-environment-setup/maven-projects/maven-simple-java-project-using-eclipse](http://www.google.com/url?q=http%3A%2F%2Fwww.raystec.com%2FHome%2Fjava-development-environment-setup%2Fmaven-projects%2Fmaven-simple-java-project-using-eclipse&sa=D&sntz=1&usg=AFQjCNFlmC6wOvoHJ9eYhKIffcCJ4w1YHQ) to setup a maven java project.

Step 2:

Add following dependencies in [pom.xml](https://sites.google.com/site/corporatejavac1/source-code-new/pom-xml)

<properties>

<springframework.version>4.0.6.RELEASE</springframework.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>**spring-core**</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>**spring-context**</artifactId>

<version>${springframework.version}</version>

</dependency>

<!-- Dependency for @PostConstruct and @PreDestroy annotations -->

<dependency>

<groupId>javax.annotation</groupId>

<artifactId>**jsr250-api**</artifactId>

<version>1.0</version>

</dependency>

</dependencies>

Dependency spring-core includes spring core jars. Spring core library jars contain org.springframework.beans and org.springframework.core packages. These packages contains BeanFactory and other core dependencies.

Dependency spring-context includes jars for ApplicationContext IOC container.

Dependency jsr250-api includes jars for JSR annotations. JSR annotations @PostConstruct and @PreDestroy are used to define bean life cycle methods

Step 3

Create User.java POJO class that contains private attributes and their setter and getter methods.

public class [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) {

private String firstName = null;

private String lastName = null;

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

}

Step 3

Create [applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-code-new/applicationcontext-xml) file and configure bean

<bean id="**userBean**" class="[com.sunilos.book.spring.bean.User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java)">

<property name="**firstName**" value="**SunilOS**" />

</bean>

Step 4

Create Test program, instantiate container and get the User bean.

public class [TestBeanFactory](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testbeanfactory) {

public static void main(String[] args) {

// Create container

BeanFactory factory = new XmlBeanFactory(new ClassPathResource("[applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-code-new/applicationcontext-xml)"));

// get User bean

// User bean = (User) context.getBean("userBean");

[User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) bean = factory.getBean("userBean", User.class);

System.out.println("Bean contains " + factory.containsBean("userBean"));

System.out.println("Type of bean " + factory.getType("userBean"));

System.out.println("is Singlton " + factory.isSingleton("userBean"));

System.out.println(bean.getFirstName());

}

}

Congratulations your first bean managed by spring IOC container.

## FAQ

1. Which maven dependencies are required for spring core?
2. Why do you include jsr250-api dependency?
3. What is the location of applicationContext.xml?

# IOC Container

**The core of the Spring Framework is its Inversion of Control (IoC) Container. Container is the factory of all java objects used in your applications.The IoC container manages life-cycle of java objects from instantiation to destruction.** The container creates objects, wires them together, configures them, and manages their complete lifecycle from creation till destruction.

There are two key responsibilities of container:

1. **Manage life-cycle of bean**
2. **Dependency injection of beans**

Java components instantiated by IoC container are called beans (Java beans). Container uses dependency injection (DI) to wire or assemble the beans that build your Application.

Spring IoC is provides loose-coupling and dynamic binding between objects dependencies. To achieve loose coupling and dynamic binding Spring IOC container injects object dependencies and gives you the ready to use object

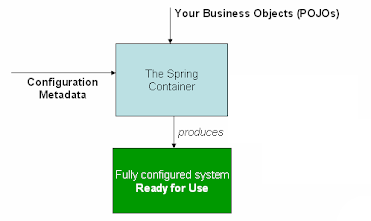
.

Container also manages bean scope, lifecycle events, and AOP features for which bean has been configured and developed.

**---------------------------------------------------------------------------------------------**

**Spring IoC container classes are part of org.springframework.beans and org.springframework.context packages**.

**BeanFactory is the root interface of Spring IoC container**. **ApplicationContext is the child interface of BeanFactory interface that provide Spring AOP features, i18n etc.**

[](https://www.google.com/url?q=https://docs.spring.io/spring/docs/current/spring-framework-reference/images/container-magic.png&sa=D&sntz=1&usg=AFQjCNGZ3FIQQqceHVv2Kel7tt8ttAD5iQ)

There are two types of IOC containers:

**[1] Spring BeanFactory Container**

This is the simplest container providing basic support for Dependency Injection (DI) and defined by the org.springframework.beans.factory.BeanFactory interface.

It is light weight and mostly used in standalone spring applications.

BeanFactory can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.

BeanFactory factory = new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));

User dto = (User) factory.getBean("user");

See: [TestBeanFactory.java](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testbeanfactory)

**[2] Spring ApplicationContext Container**

**It is child of BeanFactory interface and adds more functionalities for enterprise applications such as i18n, AOP, event publishing, web context etc**.

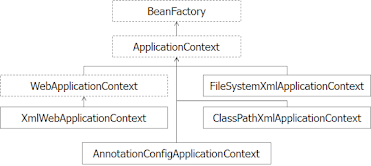
This container is defined by the org.springframework.context.ApplicationContext interface. It is recommended over the BeanFactory Enterprise applications and web applications use application context container.

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

User dto = (User) context.getBean("user");

See: [TestApplicationContext.java](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testapplicationcontext-java)

ApplicationContext has number of useful implementation classes that we can use to get spring context and then beans.



* [**AnnotationConfigApplicationContext**](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testjavaconfig-java): It is used in **standalone** application where beans are configured using annotations. If you have made standalone java applications and using annotations for configuration, then we can use this container.
* [**ClassPathXmlApplicationContext**](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testapplicationcontextxmlconfig-java): It is used in standalone application where beans are configured using XML configuration. If you have bean configuration xml file in standalone application, then you can use this.
* **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
* **AnnotationConfigWebApplicationContext** and XmlWebApplicationContext are used in web applications.
* **XmlWebApplicationContext** is used in Spring MVC web application.

## FAQ:

1. What are the differences between BeanFactory and ApplicationContext containers?
2. What are different scopes of a bean?
3. What are the differences between singleton and prototype scopes?

# Spring Beans

**Spring Bean is an object in the Spring framework that is initialized by spring container**. A Java POJO class can be a spring bean if it is configured by Spring container. Configuration information can be given by XML, annotation or Java code.

## Spring Bean Scopes

There are five scopes of a bean in spring:

1. **Singleton**: Only one instance of the bean will be created for each container. This is the default scope for a bean. Singleton bean should not have have shared instance variables, that may lead data inconsistency.
2. **Prototype**: always a new instance is created when context.getBean()method is called.
3. **Request**: This is same as prototype scope, however it is used in web applications. A new instance of the bean is created and associated with HTTP request object.
4. **Session**: A new bean is created and associated with HTTP session object in the web application.
5. **Global session**: This is used to create global session beans for Portlet applications.

Spring Framework is extendable and you can create your own custom scope. However custom scope is created in rare cases existing scopes are enough for an enterprise application. .

## Spring Bean Configuration

Spring Framework provide three ways to configure beans:

* [**XML Based Configuration**](https://sites.google.com/site/corporatejavac1/source-code-new/applicationcontext-xml)**:** beans are defined by XML tags.
* [**Annotation Based Configuration**](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testapplicationcontextannotationconfig-java)**:** beans are defined by @Service, @Repository or @Component annotations. Scope of a bean can be defined by @Scope annotation. It is started from spring 2.5.
* [**Java Based Configuration**](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testjavaconfig-java)**:** From Spring 3.0 onward beans can be defined by java programs. Key annotation are used in java configuration are @Configuration, @ComponentScan and @Bean.

### 1) XML based configuration

In this configuration beans are defined using xml tags in [applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-code-new/applicationcontext-xml) file. XML file name can be changed as per your convenience. **In Spring MVC dispature-servlet.xml file is used to configure beans.**

Beans are configured using xml <bean> tag. Here we have defined two beans Role and User.

<bean id="**roleBean**" class="[com.sunilos.book.spring.bean.Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role)" />

**<bean id="userBean" class="**[**com.sunilos.book.spring.bean.User**](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java)**">**

<property name="**firstName**" value="**SunilOS**" />

**<!--Inject role bean using setter injection -->**

<property name="**role**" ref="**roleBean**" /> // roleBean get injected.

**<!--Inject role bean using sconstructor injection -->**

<constructor-arg name="**role**" ref="**roleBean** "></constructor-arg>

**</bean>**

Dependency Injection

Reference of one bean can be injected into another bean using <property> tag. We have injected Role bean object into User bean using <property> tag:

<property name="role" ref="roleBean" />

Dependency injection can be done by two ways

(1) Setter Injection

(2) Constructor injection.

**Setter Injection**

In setter injection dependent bean is injected by setter method. Here [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) class has setter method setRole() to set the [Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role) bean. Since role bean is injected using setter method in user bean that is why it is called setter injection.

public class [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) {

...

private Role role = null;

public void setRole(Role role) {

this.role = role;

}

}

<property name="role" ref="roleBean" />

**Constructor injection**

Constructor injection is done by constructor of a bean using <constructor-arg> xml tag.

Here is example of account bean and customer bean. Account bean is injected into customer bean using constructor of [Customer](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/customer-java) class.

public class Customer {

private Account account = null;

**public Customer(Account account) {**

**this.account = account;**

**}**

}

<bean id="**accountBean**" class="[com.sunilos.book.spring.bean.Account](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/account-java)" />

<bean id="customerBean" class="[com.sunilos.book.spring.bean.Customer](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/customer-java)">

<constructor-arg name="**account**" ref="**accountBean**"></constructor-arg>

</bean>

**Life Cycle methods**

A bean can define void and zero parameter methods as life-cycle methods of a bean. Attributes init-method and destroy-method of bean xml tag are used to define life-cycle methods of a bean.

Here is bean [DataCache](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/datacache-java) that has two life cycle methods init() and clean(). Both life cycle methods are called by container when bean is instantiated and destroyed. Life cycle methods are configured using tag attributes init-method and destroy-method.

public class [DataCache](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/datacache-java){

private HashMap data = null;

..

public void init() {

data = new HashMap();

}

public void clean() {

data = null;

}

}

<bean id="dc" class="[com.sunilos.book.spring.bean.DataCache](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/datacache-java)" **init-method="init"** **destroy-method="clean"** />

**Define scopes**

Scope of a bean is defined by scope attribute of <bean> tag. By defining scope you can create singleton beans. Bean has five scopes; singleton, prototype, request, session and globalsession. Default scope is singleton.

<bean id="emailService" **scope="singleton"** class="com.sunilos.book.spring.bean.EmailService" />

Singleton beans has only one instance in its lifetime whereas instance of prototype bean is created again and again when you call getBean() method of container. Rest three scopes are used in web applications.

Factory method

ServiceLocator and DAOFactory like singleton and factory classes can not be instantiated directly. They have getInstance() factory method to instantiate them. Ideally these classes are not used in spring framework because spring itself provides container factory. But in any case you need to create instance of such classes in spring then factory-method attribute of <bean> tag is used to instantiate these classes.

public class ServiceLocator {

private ServiceLocator() {

}

private static ServiceLocator serviceLocator = null;

/\*\*

\* Get the instance of service locator

\*/

**public static ServiceLocator getInstance(){**

if (serviceLocator == null) {

serviceLocator = new ServiceLocator();

}

return serviceLocator;

}

public UserService getUserService() {

return new UserService();

}

}

<bean id="sl" class="com.sunilos.book.spring.bean.ServiceLocator" **factory-method="getInstance"** />

ServiceLocator locator = (ServiceLocator) context.getBean("sl");

### 2) Annotation based configuration

Instead of xml tags, annotations are used to define a bean.

1. Beans are defined using @Component, @Service, or @Repository annotations.
2. Annotation @Value is used to initialize attributes.
3. Scope of a bean can be defined by @Scope annotation.
4. Life cycle methods are defined by @PostConstruct and @PreDestroy annotations.
5. Dependency can be injected using @Autowired annotation.

**Annotated classes are scanned and registered in container using <context:component-scan> tag in** [**applicationContextAnotation.xml**](https://sites.google.com/site/corporatejavac1/source-code-new/applicationcontextannotation-xml) **file. :** Scan tag scanned all annotated classes from base and its sub packages and register with container.

**If you do not use <context:component-scan> tag then you have to use <annotation-driven > tag to enable annotations.**

We have example class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) that is annotated by @Component annotation. It has life cycle methods init() and destroy() both are annotated by @PostConstruct and @PreDestroy. Order has dependency of [Payment](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/payment-java) and [Inventory](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/inventory-java) bean both are injected by @Autowired annotation.

@Component("order")

@Scope("prototype")

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

@Autowired

Payment p = null;

@Autowired

Inventory i = null;

private int id = 0;

private double amount = 0;

...

@PostConstruct

public void init(){

System.out.println("This is order init method");

}

@PreDestroy

public void destroy(){

System.out.println("This is order destroy method");

}

}

//Payment bean

@Component

public class [Payment](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/payment-java) {

@Value(value = "100")

double balance = 0;

...

}

Inventory bean

@Component("inventory")

public class [Inventory](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/inventory-java){

/\*\*

\* Initialize stock by value 100

\*/

@Value(value = "100")

int stock = 0;

...

}

//Test the bean

public class [TestApplicationContextAnnotationConfig](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testapplicationcontextannotationconfig-java) {

public static void main(String[] args) {

// Initialize container

ApplicationContext context = new ClassPathXmlApplicationContext("[applicationContextAnnotation.xml](https://sites.google.com/site/corporatejavac1/source-code-new/applicationcontextannotation-xml)");

Order o = (Order) context.getBean("order");

o.placeOrder(2, 5);

System.out.println(o.getAmount());

..

}

Annotations

**1) @Component annotation**

The @Component annotation is used to define and register a java class as a bean with spring container. Here payment and user services classes are annotated by @Component to define as a bean.

@Component

public class Payment{

}

@Component

public class UserService{

}

By default name of class is used to register and identify a bean in container. First character of class name is converted to lowercase to name the bean. Payment and UserService classes will be registered by “payment” and “userService” names respectively.

You can retrieve beans from contain by:

public class [TestApplicationContextAnnotationConfig](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testapplicationcontextannotationconfig-java) {

public static void main(String[] args) {

// Initialize container

ApplicationContext context = new ClassPathXmlApplicationContext("[applicationContextAnnotation.xml](https://sites.google.com/site/corporatejavac1/source-code-new/applicationcontextannotation-xml)");

System.out.println("------Payment bean--------");

[Payment](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/payment-java) p = context.getBean("payment", Payment.class);

System.out.println("Payment balance: " + p.getBalance());

System.out.println("------User Service bean--------");

UserService service = (UserService) context.getBean("userService");

System.out.println("Authenticate :" + service.authenticate("SunilOS", "sunilOS"));

}

}

A custom bean name can be given as parameter to @Component annotation.

Here name "orderBean" is given to order bean:

@Component("**orderBean**")

public class Order {

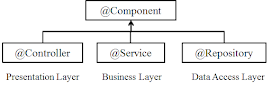
...

}

Order o = context.getBean("**orderBean** ", Order.class);

@Component specialized classes

The @Component annotation has three specialized annotations @ Controller, @Service and @Repository.



The @Controller is used to define controller classes in Spring MVC. @Service annotation is used to create business logic classes. @Repository is used to define data access objects.

**2) @Autowired annotation**

* It is used to inject bean dependencies.
* It can be applied on attribute, setter method and constructor.
* It identifies and resolves bean dependencies using bean type or bean name. @Qualifier annotation is used along with it to identify a bean by name.
* By default it identifies bean by type and injects dependencies
* When applied in constructor then it always resolves dependencies by type
* @Autowired with (required=false) is used to optionally inject a dependency.

Here is example of Order, Payment and Inventory beans. Inventory bean is injected into Order by name and Payment bean is injected using by type.

@Component("order")

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

**//identifies bean by type**

**@Autowired**

[Payment](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/payment-java) p = null;

**//identifies bean by name**

**@Autowired**

@Qualifier("**inventory**")

[Inventory](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/inventory-java) i = null;

**//identifies beans by type when applied on constructor**

**@Autowired**

public Order(Payment p,Inventory i){

this.p = p;

this.i = i;

}

...

}

**3) @Qualifier annotation**

The @Qualifier annotation is used along with @Autowired annotation to inject a dependency by name. There may be situation when you have configured one bean with two different names and you want to identify bean by name at the time of dependency injection. In this situation you will use @Qualifier annotation to identify bean by name and inject.

Here in example we have defined bean Inventory and given its name "inventory" by passing bean name in component tag. When we injecting inventory bean in order bean we have used qualifier annotation. We identified inventory bean by name by passing name "inventory" to qualifier annotation.

**@Component("inventory")**

public class Inventory {

}

**@Component("order")**

public class Order {

//Auto-wring by Name

@Autowired

**@Qualifier("inventory")**

Inventory i = null;

}

**4) @Required annotation**

@Required annotation is applied on setter methods. It indicates that the bean property must be initialized in XML configuration file at configuration time. Otherwise container throws a BeanInitializationException exception.

public class User{

private String firstName = 0;

**@Required**

**public void setFirstName(String fn) {**

**this.firstName = fn;**

**}**

..

}

<bean id="order" class="com.sunilos.book.spring.bean.Order">

<property name="**firstName**" value="**SunilOS**" />

</bean>

**5) @PostConstruct and @PreDestroy Annotations**

These are used to define bean life cycle methods

**6) @Value Annotations**

The @Value annotation is used to set default value of an attribute.

@Value(value = "100")

int stock = 0;

### 3) Java based configuration

**Instead of xml file you can use Java class to configure and auto scan spring beans**. Spring 3.0 onward beans can be defined by java programs.

Key annotation used in java configuration are @Configuration, @ComponentScan, @Bean, @Import and @DependsOn.

1. @Bean annotation is used to define a bean. It is applied on a method. Method returns a bean, this bean is registered in spring container.
2. @Configuration annotation indicates that the class will be used to define beans. Spring container will use this class as source of bean definitions.
3. @ComponentScan annotation is used to auto scan annotated components.

Here is example java program that configures java beans.

@Configuration

@ComponentScan(basePackages = { "com.sunilos.book.spring" })

public class [AppConfig](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/appconfig-java) {

@Bean(name = "userBean")

public User user() {

User user = new User();

user.setFirstName("SunilOS");// Initialize value

user.setRole(role());// Inject dependency

return user;

}

@Bean(name = "roleBean")

public Role role() {

Role role = new Role();

return role;

}

@Bean(name = "accountBean")

public Account account() {

Account account = new Account();

return account;

}

@Bean(name = "customerBean")

public Customer customer() {

Customer customer = new Customer();

customer.setAccount(account());//Inject dependency

return customer;

}

}

AnnotationConfigApplicationContext is used to read java configuration and get the container.

public class [TestJavaConfig](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-test/testjavaconfig-java) {

public static void main(String[] args) {

**AnnotationConfigApplicationContext** context = new AnnotationConfigApplicationContext();

context.register(**[AppConfig.class](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/appconfig-java)**);

context.refresh();

[User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) bean = context.getBean("userBean", User.class);

System.out.println("Bean contains " + context.containsBean("userBean"));

System.out.println("Type of bean " + context.getType("userBean"));

System.out.println("is Singlton " + context.isSingleton("userBean"));

// print first name value

System.out.println(bean.getFirstName());

System.out.println(bean.getRole().getName());

[Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) o = (Order) context.getBean("order");

o.placeOrder(2, 5);

System.out.println(o.getAmount());

System.out.println("------Customer bean--------");

[Customer](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/customer-java) customer = (Customer) context.getBean("customerBean");

System.out.println("Account balance :" + customer.getAccount().getBalance());

[UserService](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-service/userservice) service = (UserService) context.getBean("userService");

System.out.println("Authenticate :" + service.authenticate("SunilOS", "sunilOS"));

}

}

**Q: What are the difference between @Bean and @Component annotations.**

A: @Bean is used to configure a bean whereas @Component is used to define a bean.

# Dependency Injection and IOC

## Dependency Injection and Inversion of Control

An application consists of multiple objects. Multiple objects collaborate and together perform business operations in the application. Thus one object may depend on other objects to perform a business operation that is called dependency.

Here is example where three objects Order, Payment and Inventory collaborate and perform a business operation “Books a Ticket”. Order object books the order and make payment with help of Payment object. After deducting payment order updates inventory with help of Inventory object. In order to book a ticket Order requires Payment and Inventory objects thus order is depend on Payment and Inventory objects.

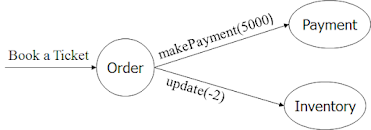


Figure: Object Dependency

### Non-IOC Dependency

Regular (non-IOC) or conventional way to resolve the dependency is to create the dependent objects and call their methods. Order has to create instances of payment and inventory classes and use them.

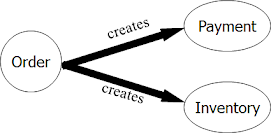


Figure: Non-IOC Dependency

Here is method placeOrder() of Order class that creates dependent objects and use them.

public class [Payment](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/payment-java) {

public double makePayment(double amt) {

balance = balance - amt;

return balance;

}

}

public class [Inventory](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/inventory-java) {

public int sold(int qty) {

stock -= qty;

return stock;

}

}

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

public void placeOrder(int qty, double price) {

**Payment p = new Payment(); //create payment object**

**Inventory i = new Inventory(); //create inventory object**

amount = qty \* price;

p.makePayment(amount);

i.sold(qty);

System.out.println("Balance: " + p.getBalance());

System.out.println("Stock : " + i.getStock());

}

}

### IOC Dependency Injection

Inversion of Control (IOC) says, now object does not need to create its dependencies, dependencies will be injected by spring IOC container using constructors or setter methods. Object defines its dependency using XML or annotations. IOC helps a developer to decouple object creation logic from business logic.

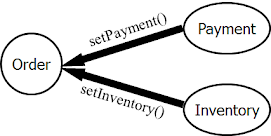


Figure: IOC Dependency Injection

Here Order defines setter injection methods. XML configuration injects Payment and Inventory objects into Order.

public class Order {

private Payment p= null;

private Inventory i= null;

public void setPayment(Payment p) {

this.p= p;

}

public void setInventory(Inventory i) {

this.i= i;

}

public void placeOrder(int qty, double price) {

amount = qty \* price;

p.makePayment(amount);

i.sold(qty);

System.out.println("Balance: " + p.getBalance());

System.out.println("Stock : " + i.getStock());

}

}

Here is metadata in applicationContext.xml file to inject dependencies in Order class.

<bean name="payment" scope="prototype" class="Payment" />

<bean name="inventory" scope="prototype" class="Inventory" />

<bean name="order" scope="prototype" class="Order">

<property name="payment" ref="payment" />

<property name="inventory" ref="inventory" />

</bean>

Now get order object from spring IOC container:

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

Order o = (Order) context.getBean("order");

o.placeOrder(2,500);

## Types of Dependency Injection

There are two types of DI.

### 1) Constructor-based dependency injection

Dependencies are injected with help of bean constructors.Constructors are defined in the bean to accept dependent objects. For example Customer bean is dependent on Address bean. In order to inject address objects one parameterized constructor is defined in Customer class. Constructor accepts shipping and billing objects of Address class. XML configuration uses <constructor-arg> tag to inject dependencies using constructor.

public class Address {

..

}

public class Customer {

private Address billingAddress = null;

private Address shippingAddress = null;

**public Customer(Address billingAddress, Address shippingAddress) {**

this.billingAddress = billingAddress;

this.shippingAddress = shippingAddress;

}

..

}

**Configuration**

<bean id="address" scope="prototype" class="com.sunilos.book.spring.bean.Address"/>

**<bean i**d="customer" scope="prototype" class="com.sunilos.book.spring.bean.Customer">

**<constructor-arg>**

**<ref bean="address" />**

**</constructor-arg>**

**<constructor-arg>**

**<ref bean="address" />**

**</constructor-arg>**

**</bean>**

### 2) Setter-based dependency injection

Dependencies can be injected with the help of setter methods. Bean has to define setter methods for dependent objects.

For example order bean is depend on inventory and payment beans. Order bean defines two setter methods, one for payment bean and second for inventory bean to inject their objects. XML configuration uses <property> tags to inject inventory and payment beans into order bean.

public class Inventory {

..

}

public class Payment {

..

}

public class Order {

private Payment payment = null;

private Inventory inventory = null;

public void setPayment(Payment payment) {

this.payment = payment;

}

public void setInventory(Inventory inventory) {

this.inventory = inventory;

}

..

}

<bean name="payment" scope="prototype" class="com.sunilos.book.spring.bean.Payment" />

<bean name="inventory" scope="prototype" class="com.sunilos.book.spring.bean.Inventory" />

<!-- Inject dependent bean references using setter injection -->

<bean name="order" scope="prototype" class="com.sunilos.book.spring.bean.Order">

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

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

</bean>

**Q: What are the differences between explicit and implicit auto-wiring ?**

A: Explicit auto-wiring is done by XML tags and implicit auto-wiring is done by **@Autowired** annotation. If both are done then **explicit auto-wiring** is considered by container.

# Bean Autowiring

## Introduction

Basic feature of spring is dependency injection. DI are done by constructors and setter methods. So far developers have to explicitly configure dependent beans in XML configuration. **Injecting the dependent beans can also be called wiring of beans**. So far developers have to mention dependencies explicitly in XML files so it was called explicit wiring. Now developers can avoid manual wiring configuration of beans because bean auto-wiring feature is available in spring.

Spring provides auto-wiring feature, that enable implicit bean dependency injection. You do not need to define bean injection details explicitly in XML configuration file.

There are two ways to configure bean auto-wiring:

**(1)XML Configuration**

Auto-writing can be enabled in XML configuration by autowire attribute in bean tag. Here we are enabling auto-wiring for [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) bean by specifying auto-wiring mode to byName.

<bean id="user" class="[com.sunilos.book.spring.bean.User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java)" **autowire="byName"** />

**(2)Annotations**

Auto writing can be done using annotation @Autowired. Annotation @Autowired can be applied on a property, setter method and constructor. Here [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) bean has annotated p and i properties by @Autowired annotation to auto-wire [Payment](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/payment-java) and [Inventory](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/inventory-java) beans.

@Component("order")

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

**@Autowired**

**Payment p = null;**

**@Autowired**

**Inventory i = null;**

...

}

## Modes of Auto wiring

Auto wiring is done as per mode values passed to it. There are multiple modes of auto-wiring:

1. autowire="**no**" : auto-writing is disabled. Bean references must be defined in the XML file via the <ref/> element (or "ref" attribute). This value is set by default.
2. autowire="**byName**" : auto-wiring is done by property name, setter method is used for dependency injection.
3. autowire="**byType**" : auto wiring is done by property data type. If data type of property is matched with a bean then bean is auto wired. Bean is auto wired if there is exactly one bean of the property type in the container. If more than one beans are found in container then a fatal error is raised.
4. autowire="**constructor**" : Just like "byType" but applied on constructor arguments. If there is not exactly one bean of the constructor argument type in the bean factory, a fatal error is raised.

Configuration of dependency using <property> and <constructor-arg> tags will override auto-wiring.

### By Name auto-wiring

**In this mode auto-wiring is done by property name. IOC container searches dependent beans by given property name and injects in required bean.**

For example [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) bean is depend on [Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role) bean. User has setRole(role) to inject the role bean dependency. IOC container looks that property name is "role" in setter method so looks up bean in container by name "role" an inject into setter method.

public class [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) {

private String firstName = null;

private String lastName = null;

private [Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role) role = null;

public void **setRole**(Role role) {

this.role = role;

}

...

}

<bean id="roleBean" class="[com.sunilos.book.spring.bean.Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role)" autowire-candidate="true" />

<bean id="user" class="[com.sunilos.book.spring.bean.User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java)" **autowire="byName"** />

Bean tag has autowire-candidate="true" attribute. Default value of autowire-candidate is true. If you set it false for a bean then bean will not participate in auto-wiring.

Since default value of autowire-candidate is true so following two statements are same:

<bean id="roleBean" class="[com.sunilos.book.spring.bean.Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role)" autowire-candidate="true" />

<bean id="roleBean" class="[com.sunilos.book.spring.bean.Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role)" />

### Auto-wiring By Type

In this mode auto-wiring is done by property datatype. IOC container searches dependent beans by property datatype and injects into required bean. Autowire attribute is set to byType in the XML configuration file. Here is following configuration container searches bean of [Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role) type and inject into User bean.

<bean id="user" class="[com.sunilos.book.spring.bean.User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java)" **autowire="byType"** />

Annotation @Autowired can be used to inject dependencies by type. @Autowired can be applied on setter, constructor or attribute.

Following example applies @Autowired to attribute p. Container will search payment bean by type and inject into order bean.

@Component("order")

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

@Autowired

[Payment](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/payment-java) p = null;

...

}

Sometimes mutiple beans of same class can be registered in container with different names. This will create confusion for IOC container to select a bean by type for dependency injection. In order to avoid this confusion @Qualifier annotation is used along with @Autowired. @Qualifier accepts bean name to be injected.

Here in example inventory bean is searched in container by name "inventory" and injected to i property.

@Component("order")

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

@Autowired

Payment p = null;

**@Autowired**

**@Qualifier("inventory")**

**Inventory i = null;**

...

}

### Auto-wiring By Constructor

It is similar to byType injection. Dependencies are injected by constructors. Beans are identified by types of constructor arguments and injected.

public class [User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java) {

public **User**(Role role) {

this.role = role;

}

...

}

<bean id="user" class="[com.sunilos.book.spring.bean.User](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/user-java)" **autowire="constructor"** />

Annotation @Autowired can be applied on constructors to inject dependencies.

@Component("order")

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

Payment p = null;

Inventory i = null;

@Autowired

public **Order**(Payment p, Inventory i) {

this.p= p;

this.i= i;

}

...

}

@Qualifier annotation can be applied on constructor arguments to identify a bean by name from container.

public class [Order](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/order-java) {

Payment p = null;

Inventory i = null;

@Autowired

public **Order**(@Qualifier("payment") Payment p, @Qualifier("inventory") Inventory i) {

this.p= p;

this.i= i;

}

...

}

### Excluding a bean from auto-wiring

You can exclude a bean from auto-wiring. It is done by setting value of autowire-candidate attribute of <bean> tag to false.

<bean id="roleBean" class="[com.sunilos.book.spring.bean.Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role)" autowire-candidate="**false**" />

### Creating a primary bean

When we define two or more beans of the same type, and these beans are get autowired by byType in some other bean, then an ambiguity arises. To overcome these ambiguities with autowiring by type, spring offers another option, creation of a primary bean. We can create a primary bean by setting the primary attribute of <bean> tag to true. For example,

<bean id="role" scope="prototype" class="[com.sunilos.book.spring.bean.Role](https://sites.google.com/site/corporatejavac1/source-code-new/com-sunilos-book-spring-bean/role)" **primary="true"** />

By setting the primary attribute of <bean> tag to true, the bean becomes a primary bean and the bean gets autowired with the calling bean. But when we declare more than one bean as primary, then a conflict comes and an exception arises.

## Auto-wiring By Annotation

You can use annotation @Autowired to autowire a bean. Annotation can be applied on attributes, setter methods and constructors. By default it resolves dependency by Type, @Qualifier annotation can be used along with annotation to resolve dependencies by Name.

Spring 3 provides three annotations for autowiring:

1. Spring’s own @Autowired annotation.
2. The @Inject annotation (JSR-330).
3. The @Resource annotation (JSR-250).

Annotation based configuration is enabled by <context:annotation-config /> tag is applicationContext.xml file.

## Limitations of auto-wiring

1. Both <property> and <constructor-arg> tags always override auto-wiring.
2. Wiring information may not be visible outside of the container. for example to some documentation tools.
3. If there are several beans of a particular type exist, a no unique bean exception occurs.

## FAQ

1. What is auto-wiring and how you can do it?
2. How many ways are there to configure auto-wiring?
3. Which are the different modes of auto-wiring?
4. What are the differences between byName and byType auo-wiring?
5. What is constructor auto-wiring?
6. Which auto wiring mode is used by @Autowired annotation?
7. Can you apply @Autowired annotation on an attribute?
8. Can you apply @Autowired annotation on a method?
9. Can you apply @Autowired annotation on a constructor?
10. Why do you use @Qualifier annotation along with @Autowired?
11. How you can exclude a bean from auto-wiring?
12. If we have applied both implicit (@Autowired) and explicit wiring ( <property> and <constructor-arg> ) then which one will be applied?

# Auto Discovery

Dependency injection can be done by xml configuration or auto-wiring. Auto wiring significantly minimizes size of XML file.

In order to further reduce XML configuration you can define beans with following annotations:

1. @Component: creates simple beans
2. @Repository: creates DAOs
3. @Service: creates Services
4. @Controller: creates controllers

These annotated beans and their dependencies can be automatically discovered <context:component-scan tag in applicationContext.xml file. With help of auto-discovery you do not need to manually register the beans in spring config file.

Here is sample configuration that auto discovers beans from com.sunilos.book.spring package.

<context:component-scan base-package="com.sunilos.book.spring" />

# Spring DAO

## Introduction

The Spring-DAO ( Data Access Object ) provides a consistent way to use data access frameworks like JDBC, Hibernate, JPA, JDO and other ORM frameworks.

It provides consistent exception hierarchy. This makes your application independent from underlying data access framework. You can provide consistent exception handling without worrying about specific persistent technology.

There are three key features of Spring DAO:

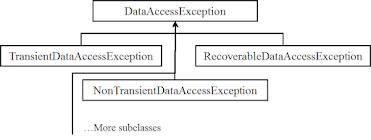
1. **Resource Management**:
   1. Now programmer does not need to open or close a database connection.
   2. Spring DAO implicitly opens and closes database connections.
   3. Programmer do not need to write statements for opening and closing the database connections.
2. **Exception translation**:
   1. It translates all native JDBC and ORM exceptions into unchecked DataAccessException.
   2. DataAccessException exception is propagate to service layer.
   3. Programmer do not need to handle database exceptions.
   4. Service classes handles database exception.
3. **Transaction handling**:
   1. Service classes handles transactions with help of Spring AOP.
   2. If unchecked exception is propagated from @Transnational method then transaction is rolled back.

## Consistent exception hierarchy

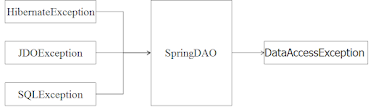
There are different data access frameworks like JDBC, Hibernate, JPA, JDO etc. They have there own native Exceptions those are raised in exceptional cases. JDBC raises SQLException, Hibernate raises HibernateException, and JDO raises JDOException.

Spring provide a generic exception class hierarchy. Spring wraps all technology specific exceptions like SQLException, JDOException and HibernateException into a generic exception class DataAccessException or its subclass.

Spring-DAO translates ORM specific exceptions into DataAccessException or into its subclasses. Spring enables transparent exception translation using @Repository annotation.



Spring makes you free from handling different exceptions when you are working on different persistent frameworks.



## Annotations used for configuring DAO or Repository classes

Annotation @Repository is used to make any POJO class to a spring data access object. This annotation triggers component scanning and repository configuring within the class definition rather than using XML configuration entries.

**@Repository**

public class [CollegeDAOJDBCImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaojdbcimpl-java) implements [CollegeDAOInt](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaoint-java) {

private JdbcTemplate jdbcTemplate;

**@Autowired**

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

A DAO requires access to a Persistent Resource (aka Data Connection Pool) to get connection to the Database. Depending on persistence technologies there are different Persistent Resource like JDBC need to access DataSource, JPA need to access EntityManager, and Hibernate need to access SessionFactory.

The easiest way to inject Persistent Resource dependency use one of the @Autowired, @Inject, @Resource, @PersistenceContext annotations.

In order to scan annotated classes you must contain following tag in your ApplicationContext.xml.

<context:component-scan base-package="com.sunilos..dao" />

### JDBC

JDBC uses DataSource as Data Connection Pool.

<!-- Data Source Definition -->

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

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

<property name="url" value="jdbc:mysql://localhost:3306/ST\_JAVA" />

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

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

</bean>

JDBC DAO is created using @repository annotation and injecting data-source into dao. JDBCTemplate object is created from data-source.

@Repository

public class [CollegeDAOJDBCImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaojdbcimpl-java) implements [CollegeDAOInt](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaoint-java) {

private JdbcTemplate jdbcTemplate;

**@Autowired**

**public void setDataSource(DataSource dataSource) {**

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

### JPA

JPA uses @PersistenceContext annotation to inject Data Connection Pool.

@Repository

public class JPADAOImpl implements JPADAOInt {

**@PersistenceContext**

private EntityManager entityManager;

// ...

}

### Hibernate

Hibernate has SessionFactory as Data Connection Pool (Persistent Resource). Hibernate DAO is created using @repository annotation and injecting session factory into dao.

@Repository

public class [CollegeDAOHibImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaohibimpl) implements [CollegeDAOInt](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaoint-java) {

**@Autowired**

**private SessionFactory sessionFactory;**

//...

# Spring JDBC

## Introduction

Spring provides an abstract way to communicate with database using plan old JDBC objects. Spring JDBC Framework takes care of all the low-level details to open the connection, prepare and execute the SQL statements, process exceptions, handle transactions and finally close the connection. You just have to define connection parameters and specify the SQL statements to be executed, fetch the query results and iterate them.

Spring-JDBC provides several approaches and correspondingly classes to communicate with database. The most popular approach is to use **JdbcTemplate** class to communicate with database. JdbcTemplate is the central framework class that manages all the database communication and exception handling. All approaches require JDBC 2.0 and above drivers

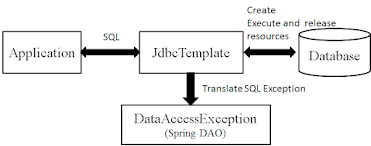
The org.springframework.jdbc.core package contains the JdbcTemplate class and its various callback interfaces, plus a variety of related classes. A subpackage named org.springframework.jdbc.core.simple contains the SimpleJdbcInsert and SimpleJdbcCall classes. Another subpackage named org.springframework.jdbc.core.namedparam contains the NamedParameterJdbcTemplate class and the related support classes.

## The JdbcTemplate

It is the basic thread-safe class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection. It performs the basic tasks of the core JDBC workflow such as statement creation and execution, leaving application code to provide SQL and extract results.

The JdbcTemplate class

* executes SQL queries,
* executes Update and Insert statements
* executes Stored Procedure calls,
* Performs iteration over ResultSets and extraction of returned parameter values.
* It catches JDBC exceptions and translates them to the generic, more informative, DataAccessException exception hierarchy defined in the Spring DAO.



### Execute SELECT Query

You can execute different type of SELECT queries with help of JdbcTemplate. Here we can see few examples:

**Get count to total number of records from PART Table :**

int rowCount = jdbcTemplate.queryForObject("select count(\*) from PART", Integer.class);

Method queryForObject receives two parameters. First parameter is sql query. Second parameter is class type in which query result is typecasted.

**A simple query using a bind variable: get number of PARTS, produced in Unit number #2.**

int rowCount = jdbcTemplate.queryForObject( "select count(\*) from PART where UNIT\_ID = ?", Integer.class, 2);

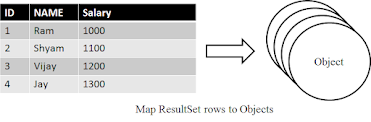
Here method queryForObject receives 3 parameters. First parameter is sql query. Second parameter is class type in which query result is typecasted. Third parameter is value of ? in sql query.

**RowMapper**

It is an interface found in org.springframework.jdbc.core package. Its implementation class is used by JdbcTemplate to map a row of ResultSet into an Object.

RowMapper objects are stateless and thus reusable. It is used either by query methods of JdbcTemplate or by OUT parameters of stored procedures.

All exception raised buy RowMapper object will be handled by calling JdbcTemplate.



Here is an example code of RowMapper class that converts ResultSet into College object.

public class [CollegeMapper](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegemapper-java) implements **RowMapper<College>** {

**public College mapRow(ResultSet rs, int args) throws SQLException {**

[College](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-dto/college-java) dto = new [College](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-dto/college-java)();

dto.setId(rs.getLong("id"));

dto.setName(rs.getString("name"));

dto.setAddress(rs.getString("address"));

//...

return dto;

}

}

**Get list of colleges from COLLEGE table using RowMapper object.**

List<College> colleges = jdbcTemplate.query("SELECT \* FROM ST\_COLLEGE",**new CollegeMapper()**);

### Execute INSERT

String sql = "INSERT INTO ST\_COLLEGE VALUES(?,?,?)";

jdbcTemplate.update(sql, 1, "IPS", "Rau");

### Execute UPDATE

String sql = "UPDATE ST\_COLLEGE SET NAME=?,ADDRESS=? WHERE ID=?";

jdbcTemplate.update(sql, "Medicaps", "Rau",1);

### Execute DELETE

String sql = "delete from ST\_COLLEGE where id = ?";

int i = jdbcTemplate.update(sql, 1);

### Execute DDL Statements

Method execute() can be used to execute any arbitrary SQL. It is typically used to execute DDL statements, Stored Procedures and Stored Functions.

Here is an example of calling DDL Statement.

jdbcTemplate.execute("create table ACCOUNT (id integer, name varchar(100))");

### Execute Stored Procedure and Stored Function

Here is an example of calling Stored Procedure.

jdbcTemplate.update("call USER\_COUNT(?)", 25L);

### JDBCTemplate best practices.

Instances of the JdbcTemplate class are thread-safe once configured. This is important because it means that you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs (or repositories). The JdbcTemplate is stateful, it contains a reference of a DataSource, but this state is not conversational state.

A common practice when using the JdbcTemplate class ( and the associated NamedParameterJdbcTemplate classes) is to configure a DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes; the JdbcTemplate is created in the setter for the DataSource. Here is the sample code of best practice.

**@Repository**

public class [CollegeDAOJDBCImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaojdbcimpl-java) implements [CollegeDAOInt](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaoint-java) {

private JdbcTemplate jdbcTemplate;

**@Autowired**

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = **new JdbcTemplate(dataSource);**

}

## Example of CRUD Operations

### Native Approach (Recommended)

Recommended approach is to inject the Datasource and create JdbcTemplate object in the DAO WITHOUT inheriting JdbcDaoSuppot class.

**Step #1 :** Create College POJO class that's object will represent a record in COLLEGE table.

public class [College](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-dto/college-java){

protected long id;

//...

}

**Step #2 :** Create RowMapper class that will map a record from COLLEGE table to College object.

public class [CollegeMapper](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegemapper-java) implements **RowMapper<College>** {

**public College mapRow(ResultSet rs, int args) throws SQLException {**

[College](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-dto/college-java) dto = new [College](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-dto/college-java)();

dto.setId(rs.getLong("id"));

//...

return dto;

}

}

**Step #3 :** Create interface CollegeDAOInt that contains CRUD methods.

public interface [CollegeDAOInt](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaoint-java) {

public long add(College dto);

public long update(College dto);

public College delete(long id);

public College findByName(String name);

public College findByPK(long pk);

public List search(College dto);

}

**Step #4:** Create class CollegeDAOImpl that will implement CollegeInt interface.

**@Repository**

public class [CollegeDAOJDBCImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaojdbcimpl-java) implements [CollegeDAOInt](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaoint-java) {

private JdbcTemplate jdbcTemplate;

**@Autowired**

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

**Step #5:** Configure datasource in [applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-jdbc-xml).

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

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

<property name="url" value="jdbc:mysql://localhost:3306/ST\_JAVA" />

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

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

</bean>

**Step #6:** Test bean

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

CollegeDAOInt dao = (CollegeDAOInt) context.getBean("**collegeDAO**");

College dto = new College();

dto.setId(1L);

dto.setName("Vijay");

dto.setAddress("Mumbai");

dto.setCity("Indore");

dto.setState("MP");

dto.setPhoneNo("99999999");

**dao.add(dto);**

### Classic Approach

Classic approach inherits DAOSupport classes. This approach is NOT recommended anymore.

Here is Account example class that implements DAO by inheriting JdbcDaoSupport support class.

public class [AccountDAOJDBCImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-dao/classic/accountdaojdbcimpl-java) extends **JdbcDaoSupport** implements AccountDAOInt {

public long add(Account ac) {

String sql = "insert into ST\_ACCOUNT (ID, TYPE, BALANCE) values (?,?,?)";

**getJdbcTemplate().**update(sql, ac.getId(), ac.getType(), ac.getBalance());

return ac.getId();

}

Following configurations will be made in applicationContext.xml to inject dao dependencies:

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

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

</bean>

<!-- Account JDBC DAO Objects -->

<bean id="**accountDAO**" class="com.sunilos.dao.classic.**AccountDAOJDBCImpl**">

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

</bean>

### Service Class

Service class contains business logic. It is defined by @Service annotation. It does transaction handling with help of Spring AOP.

There are two ways to apply transactions:

* XML Configuration
* @Transactional annotation

**XML Configuration**

Here is sample code of UserService and its respective configuration setting in [applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-jdbc-xml).

public class [UserServiceImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-service/userserviceimpl-java) implements UserServiceInt {

private UserDAOInt dao = null;

private static Logger log = Logger.getLogger(UserServiceImpl.class);

public long add(UserDTO dto) {

long pk = dao.add(dto);

return pk;

}

[applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-jdbc-xml) configuration

<bean id="userService" class="com.sunilos.service.UserServiceImpl"

autowire="byType" />

<!-- JDBC Transaction Management -->

<bean id="jdbcTransactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

<!-- The transactional advice contains methods and attribute mapping for

JDBC Transaction Manager -->

<tx:advice id="jdbcTxAdvice" transaction-manager="jdbcTransactionManager">

<tx:attributes>

<!-- all methods starting with 'find' and 'search' are read-only -->

<tx:method name="find\*" read-only="true" />

<tx:method name="search\*" read-only="true" />

<!-- other methods use the default transaction settings (see below) -->

<tx:method name="\*" propagation="REQUIRED" />

</tx:attributes>

</tx:advice>

<!-- ensure that the above transactional advice runs for any execution of

an operation defined by the \*DAOHibImpl\* classes -->

<aop:config>

<aop:pointcut id="serviceOperations"

expression="execution(\* com.sunilos.service.\*ServiceImpl.\*(..))" />

<aop:advisor advice-ref="jdbcTxAdvice" pointcut-ref="serviceOperations" />

</aop:config>

**Annotation @Transactional**

Here is sample beans College and its [applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-ann-xml) configuration that applies annotation based transaction.

@Service(value = "collegeService")

public class [CollegeServiceImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/service/collegeserviceimpl-java) implements CollegeServiceInt {

@Autowired

private CollegeDAOInt dao = null;

@Transactional(readOnly = true)

public College get(long id) {

return dao.findByPK(id);

}

@Transactional(propagation = Propagation.REQUIRED)

public long add(College dto) {

long id = dao.add(dto);

return id;

}

[applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-ann-xml) configuration

<!--Scan @Repository, @Service, @Component and @Controller spring beans -->

<context:component-scan base-package="com.sunilos.annotation" />

<!-- enable the configuration of transactional behavior based on annotations -->

<tx:annotation-driven transaction-manager="transactionManager" />

### Test case

[UserServiceTestase](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-test/userservicetestcase-java)

[CollegeServiceTestcase](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-test/collegeservicetestcase-java)

[AccountServiceTestcase](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-test/accountservicetestcase-java)

## Maven Dependency

<!-- JDBC Data Access Library (depends on spring-core, spring-beans, spring-context,

spring-tx) Define this if you use Spring's JdbcTemplate API (org.springframework.jdbc.\*) -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>3.0.5.RELEASE</version>

</dependency>

[Spring ORM](https://sites.google.com/site/corporatejavac1/spring/spring-orm)

# Spring ORM

## Introduction

Spring provides a convenient way to integrate object relational persistence frameworks such as Hibernate, JPA, and JDO for data access object, transaction and resource management implementation. Spring supports ORM integration with generic transaction features and DAO exception hierarchy.

**Advantage of Spring ORM**

* **Easier testing:** Spring’s IoC approach makes it easy to swap the implementations and configuration locations of Hibernate SessionFactory instances, JDBC DataSource instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.
* **Common data access exceptions:** Spring can wrap exceptions from your ORM tool, converting them to a common runtime DataAccessException hierarchy.
* **General resource management:** Spring application contexts can handle the location and configuration of Hibernate SessionFactory instances, JPA EntityManagerFactory instances, JDBC DataSource instances, and other related resources. This makes these values easy to manage and change.
* **Integrated transaction management:** You can wrap your ORM code with a declarative, aspect-oriented programming (AOP) style method interceptor either through the @Transactional annotation or by explicitly configuring the transaction AOP advice in an XML configuration file.

## Hibernate Integration

Hibernate objects can be defined in Spring container as a bean and injected in DAO objects.

### Session Factory Setup

Spring container is the factory of all kinds of objects. Hibernate SessionFactory can be defined in Spring IOC container as a bean. Here is XML application context definition that shows how to set up a JDBC Datasource and a Hibernate SessionFactory.

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

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

<property name="url" value="jdbc:mysql://localhost:3306/ST\_JAVA" />

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

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

</bean>

<bean id="**sessionFactory**" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

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

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">

org.hibernate.dialect.MySQLDialect

</prop>

<prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.hbm2ddl.auto">update</prop>

</props>

</property>

<property name="mappingResources">

<list>

<value>in/co/sunrays/dto/Account.hbm.xml</value>

<value>in/co/sunrays/dto/User.hbm.xml</value>

</list>

</property>

</bean>

### Plain Hibernate 3 API DAO Implementation

Hibernate 3 has a feature called contextual sessions. Hibernate itself manages one current Session per transaction in contextual sessions. This is roughly equivalent to Spring’s synchronization of one Hibernate Session per transaction. This approach is recommended to develop DAO classes in Hibernate.

Here is the DAO implementation example using plain Hibernate 3 API:

@Repository

public class [CollegeDAOHibImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaohibimpl) implements [CollegeDAOInt](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/dao/collegedaoint-java) {

@Autowired

private SessionFactory sessionFactory;

public long add(College dto) {

Session session = sessionFactory.getCurrentSession();

session.save(dto);

return dto.getId();

}

public long update(College dto) {

Session session = sessionFactory.getCurrentSession();

session.update(dto);

return dto.getId();

}

Make following entry in [ApplicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-ann-xml)

<context:component-scan base-package="com.sunilos.annotation" />

### Classic Approach

Classic approach inherits DAOSupport classes. This approach is not recommended anymore.

public class AccountDAOHibImpl extends HibernateDaoSupport implements

AccountDAOInt {

public long add(Account bean) {

getHibernateTemplate().save(bean);

return bean.getId();

}

public long update(Account bean) {

getHibernateTemplate().update(bean);

return bean.getId();

}

Make following entry in [ApplicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-hib-xml)

<bean id="accountDAO" class="com.sunilos.dao.classic.AccountDAOHibImpl">

<property name="sessionFactory" ref="sessionFactory" />

</bean>

### Service Class

Service class contains business logic. It is defined by @Service annotation. It does transaction handling with help of Spring AOP.

There are two ways to apply transactions:

* XML Configuration
* @Transactional annotation

**XML Configuration**

Here is sample code of UserService and its respective configuration setting in [applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-hib-xml).

public class [UserServiceImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-service/userserviceimpl-java) implements UserServiceInt {

private UserDAOInt dao = null;

private static Logger log = Logger.getLogger(UserServiceImpl.class);

public long add(UserDTO dto) {

long pk = dao.add(dto);

return pk;

}

[applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-hib-xml) configuration

<bean id="userService" class="com.sunilos.service.UserServiceImpl"

autowire="byType" />

<!-- Hibernate Transaction Manager -->

<bean id="hibTransactionManager"

class="org.springframework.orm.hibernate3.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory" />

</bean>

<!-- Configure Declarative Transaction on Service classes using Spring AOP -->

<!-- The transactional advice contains methods and attribute mapping for

Hibernate Transaction Manager -->

<tx:advice id="hibTxAdvice" transaction-manager="hibTransactionManager">

<tx:attributes>

<!-- all methods starting with 'find' and 'search' are read-only -->

<tx:method name="find\*" read-only="true" />

<tx:method name="search\*" read-only="true" />

<!-- other methods use the default transaction settings -->

<tx:method name="\*" propagation="REQUIRED" />

</tx:attributes>

</tx:advice>

<!-- ensure that the above transactional advice runs for any execution of

an operation defined by the \*ServiceImpl\* classes -->

<aop:config>

<aop:pointcut id="serviceOperations"

expression="execution(\* com.sunilos.service.\*ServiceImpl.\*(..))" />

<aop:advisor advice-ref="hibTxAdvice" pointcut-ref="serviceOperations" />

</aop:config>

**Annotation @Transactional**

Here is sample beans College and its [applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-ann-xml) configuration that applies annotation based transaction.

@Service(value = "collegeService")

public class [CollegeServiceImpl](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos/service/collegeserviceimpl-java) implements CollegeServiceInt {

@Autowired

private CollegeDAOInt dao = null;

@Transactional(readOnly = true)

public College get(long id) {

return dao.findByPK(id);

}

@Transactional(propagation = Propagation.REQUIRED)

public long add(College dto) {

long id = dao.add(dto);

return id;

}

[applicationContext.xml](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/applicationcontext-ann-xml) configuration

<!--Scan @Repository, @Service, @Component and @Controller spring beans -->

<context:component-scan base-package="com.sunilos.annotation" />

<!-- enable the configuration of transactional behavior based on annotations -->

<tx:annotation-driven transaction-manager="hibTransactionManager" />

### Test case

[UserServiceTestase](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-test/userservicetestcase-java)

[CollegeServiceTestcase](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-test/collegeservicetestcase-java)

[AccountServiceTestcase](https://sites.google.com/site/corporatejavac1/source-c/spring-dao-1/com-sunilos-test/accountservicetestcase-java)

## Maven Dependency

<!-- Hibernate O/RM implementation of the JPA specification -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>3.5.0-Final</version>

</dependency>

<!-- Hibernate is depends on slf4j-api and hibernate-entitymanager -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>3.5.0-Final</version>

</dependency>

# Jasper Report

## Introduction

JasperReports is the world's most popular open source reporting engine. It is entirely written in Java and it is able to use data coming from any kind of data source and produce pixel-perfect documents that can be viewed, printed or exported in a variety of document formats including HTML, PDF, Excel, OpenOffice and Word.

[http://jasperforge.org/projects/jasperreports](http://www.google.com/url?q=http%3A%2F%2Fjasperforge.org%2Fprojects%2Fjasperreports&sa=D&sntz=1&usg=AFQjCNEd3GEeWoOkLrLKnsWc1mFbkaAE-Q)

[http://en.wikipedia.org/wiki/JasperReports](http://www.google.com/url?q=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FJasperReports&sa=D&sntz=1&usg=AFQjCNHLTmUV7rUaJZ5Fczri29KFO8KkRg)

## FAQ

Q: What is JASPER Report?

A: This is a open source Reporting tool.

Q: Where do you use JASPER?

A: We use it in our Application to generate reports.

Q: What report formats can be generated by JASPER?

A: PDF, Excel, Doc, and HTML.

Q: What Editor do you use?

A: iReport.

Q: What is the extension of Jasper report file?

A: Extension is .jasper.

Q: What is the extension of iReport files?

A: Extension is .jrxml.

Q: Which file will be genrated when iReport compile jsxml?

A: It will generate .jasper file.

Q: How do you integrated .jasper file with JAVA code?

Q: How do you integrate .jasper file with Struts 2?

|  |
| --- |
|  |