1. What is Loose Coupling?

Ans Loose coupling is an approach to interconnecting the components in a system or [network](https://searchnetworking.techtarget.com/definition/network) so that those components, also called elements, depend on each other to the least extent practicable. Coupling refers to the degree of direct knowledge that one element has of another.

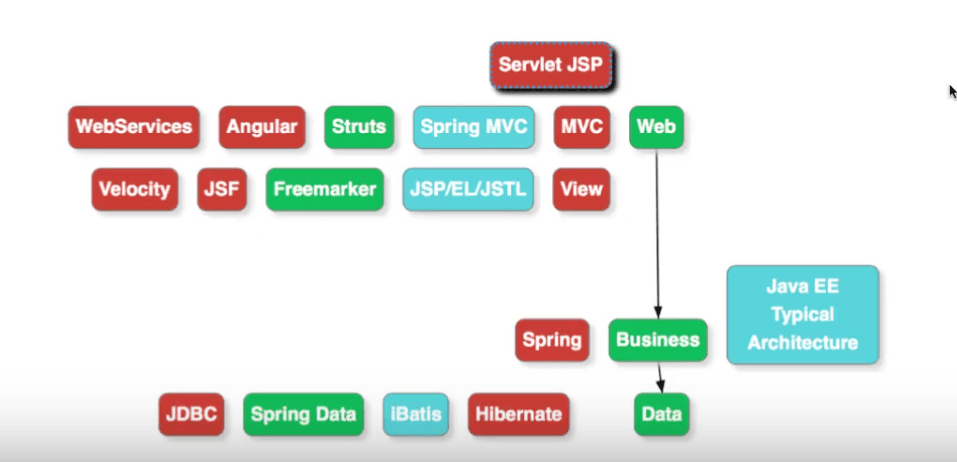
The goal of a loose coupling architecture is to reduce the risk that a change made within one element will create unanticipated changes within other elements. Limiting interconnections can help isolate problems when things go wrong and simplify testing, maintenance and troubleshooting procedures.

A loosely coupled system can be easily broken down into definable elements. The extent of coupling in a system can be measured by mapping the maximum number of element changes that can occur without adverse effects. Examples of such changes include adding elements, removing elements, renaming elements, reconfiguring elements, modifying internal element characteristics and rearranging the way in which elements are interconnected.

Although loose coupling minimizes unnecessary interaction among system elements, it can create problems when such interaction is desired. For example, in some data-centric systems a high degree of element interdependence is necessary for synchronization in real time.

1. What is a Dependency?

**Dependencies At A High Level**

We build enterprise applications in multiple layers: 

A typical Java application will have three layers in its architecture: web, business and data.

* The web layer
* The business layer
* The data layer

In the above scenario:

* Web Layer depends on Business Layer. The business layer is a dependency for the web layer.
* Business layer depends on Data Layer. The data layer is a dependency for the business layer.

**Dependencies At Class Level**

Let’s look at an example:

@Service

public class ClientBOImpl implemented ClientBO {

@Autowired

ProductDO productDO;

@Autowired

ClientDO clientDo;

@Override

public Amount getClientProductsSum(long cliendId) {

*//...*

}

@Override

public void saveChangedProducts(long clientId,

List<Product> userEnteredProducts) {

*//...*

}

*//...*

}

ClientBOImpl is the business class, and it makes use of two data layer classes - ProductDO and ClientDO.

Let’s now have a look at the business logic within ````ClientBOImpl```:

* getClientProductsSum() : This returns the sum of all products for a given client.
* saveChangedProducts() : When products are modified on the application page, this method is called.

Both methods in ClientBOImpl need either ProductDO or ClientDO. ProductDO and ClientDO are dependencies of ClientBOImpl.

**Inputs/Outputs Are Not Dependencies**

If you look at public Amount getClientProductsSum(long clientId), clientId is merely an input, not a dependency. Similarly, the total calculated amount returned by getClientProductsSum is an output, not a dependency.

**A Few More Examples Of Dependencies**

**Example-1**

Have a look at the following code:

@Component

public class ComplexAlgorithmImpl {

@Autowired

private SortAlgorithm sortAlgorithm;

*//...*

}

public interface SortAlgorithm {

public int[] sort(int[] numbers);

}

@Component

public class QuickSortAlgorithm implements SortAlogrithm {

*//...*

}

ComplexAlgorithmImpl performs a lot of complex logic, and sorting is one of the steps.

The SortAlgorithm is a dependency of ComplexAlgorithmImpl.

Since SortAlgorithm is an interface, you can easily change the actual sort algorithm used by ComplexAlgorithmImpl, without changing its code.

**Example-2**

Consider the following code:

import java.sql.ResultSet;

@Repository

public class PersonJdbcDao {

@Autowired

JdbcTemplate jdbcTemplate;

class PersonRowMapper implements RowMapper<Person> {

@Override

public Person mapRow(ResultSet rs, int rowNum) throws SQLException {

*//...*

}

*//...*

}

}

To execute a query on the database, PersonJdbcDao needs JdbcTemplate. Therefore, JdbcTemplate is a dependency of PersonJdbcDao.

Let’s look at a simple method:

public Person findById(int id) {

return jdbcTemplate.queryForObject(*//...);*

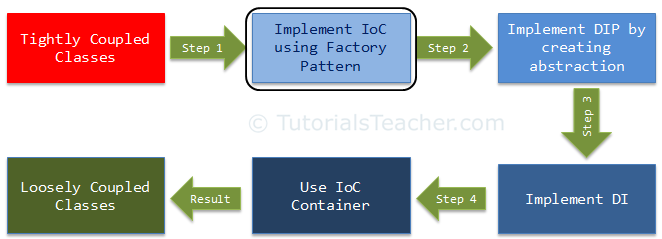
}

id is the input for this method, and the output returned is of type Person.

In the above method, we are making use of a dependeny jdbcTemplate . The inputs and outputs are not dependencies.

1. What is IOC (Inversion of Control)?

Ans 3:- This is the first step towards achieving loose coupled design, as illustrated by the following figure:

[](https://www.tutorialsteacher.com/Content/images/ioc/ioc-step1.png)

Inversion of Control (IoC) is a design principle (although, some people refer to it as a pattern). As the name suggests, it is used to invert different kinds of controls in object-oriented design to achieve loose coupling. Here, controls refer to any additional responsibilities a class has, other than its main responsibility. This include control over the flow of an application, and control over the flow of an object creation or dependent object creation and binding.

IoC is all about inverting the control. To explain this in layman's terms, suppose you drive a car to your work place. This means you control the car. The IoC principle suggests to invert the control, meaning that instead of driving the car yourself, you hire a cab, where another person will drive the car. Thus, this is called inversion of the control - from you to the cab driver. You don't have to drive a car yourself and you can let the driver do the driving so that you can focus on your main work.

The IoC principle helps in designing loosely coupled classes which make them testable, maintainable and extensible.

Let's understand how IoC inverts the different kinds of control.

### Control Over the Flow of a Program

In a typical console application in C#, execution starts from the Main() function. The Main() function controls the flow of a program or, in other words, the sequence of user interaction. Consider the following simple console program.

Example: Program Flow

namespace FlowControlDemo

{

class Program

{

static void Main(string[] args)

{

bool continueExecution = true;

do

{

Console.Write("Enter First Name:");

var firstName = Console.ReadLine();

Console.Write("Enter Last Name:");

var lastName = Console.ReadLine();

Console.Write("Do you want to save it? Y/N: ");

var wantToSave = Console.ReadLine();

if (wantToSave.ToUpper() == "Y")

SaveToDB(firstName, lastName);

Console.Write("Do you want to exit? Y/N: ");

var wantToExit = Console.ReadLine();

if (wantToExit.ToUpper() == "Y")

continueExecution = false;

}while (continueExecution);

}

private static void SaveToDB(string firstName, string lastName)

{

//save firstName and lastName to the database here..

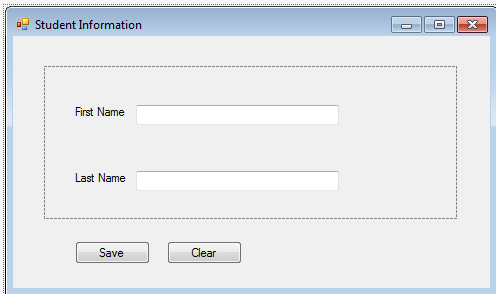
}

}

}

In the above example, the Main() function of the program class controls the flow of a program. It takes the user's input for the first name and last name. It saves the data, and continues or exits the console, depending upon the user's input. So here, the flow is controlled through the Main() function.

IoC can be applied to the above program by creating a GUI-based application such as the following windows-based application, wherein the framework will handle the flow of a program by using events.

[](https://www.tutorialsteacher.com/Content/images/ioc/winform.png)

This is a simple example of implementing IoC in the flow of a program.

### Control Over the Dependent Object Creation

IoC can also be applied when we create objects of a dependent class. First of all, let's understand what we mean by dependency here.

Consider the following example.

public class A

{

B b;

public A()

{

b = new B();

}

public void Task1() {

// do something here..

b.SomeMethod();

// do something here..

}

}

public class B {

public void SomeMethod() {

//doing something..

}

}

In the above example, class A calls b.SomeMethod() to complete its task1. Class A cannot complete its task without class B and so you can say that "Class A is dependent on class B" or "class B is a dependency of class A".

In the object-oriented design approach, classes need to interact with each other in order to complete one or more functionalities of an application, such as in the above example - classes A and B. Class A creates and manages the life time of an object of class B. Essentially, it controls the creation and life time of objects of the dependency class.

The IoC principle suggests to invert the control. This means to delegate the control to another class. In other words, invert the dependency creation control from class A to another class, as shown below.

public class A

{

B b;

public A()

{

b = Factory.GetObjectOfB ();

}

public void Task1() {

// do something here..

b.SomeMethod();

// do something here..

}

}

public class Factory

{

public static B GetObjectOfB()

{

return new B();

}

}

As you can see above, class A uses Factory class to get an object of class B. Thus, we have inverted the dependent object creation from class A to Factory. Class A no longer creates an object of class B, instead it uses the factory class to get the object of class B.

Let's understand this using a more practical example.

In an object-oriented design, classes should be designed in a loosely coupled way. Loosely coupled means changes in one class should not force other classes to change, so the whole application can become maintainable and extensible. Let's understand this by using typical n-tier architecture as depicted by the following figure:

[](https://www.tutorialsteacher.com/Content/images/ioc/demo-architecture.png)

In the typical n-tier architecture, the User Interface (UI) uses Service layer to retrieve or save data. The Service layer uses the BusinessLogic class to apply business rules on the data. The BusinessLogic class depends on the DataAccess class which retrieves or saves the data to the underlying database. This is simple n-tier architecture design. Let's focus on the BusinessLogic and DataAccess classes to understand IoC.

The following is an example of BusinessLogic and DataAccess classes for a customer.

public class CustomerBusinessLogic

{

DataAccess \_dataAccess;

public CustomerBusinessLogic()

{

\_dataAccess = new DataAccess();

}

public string GetCustomerName(int id)

{

return \_dataAccess.GetCustomerName(id);

}

}

public class DataAccess

{

public DataAccess()

{

}

public string GetCustomerName(int id) {

return "Dummy Customer Name"; // get it from DB in real app

}

}

As you can see in the above example, the CustomerBusinessLogic class depends on the DataAccess class. It creates an object of the DataAccess class to get the customer data.

Now, let's understand what's wrong with the above classes.

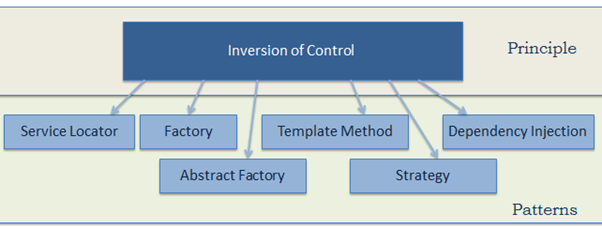
In the above example, CustomerBusinessLogic and DataAccess are tightly coupled classes because the CustomerBusinessLogic class includes the reference of the concrete DataAccess class. It also creates an object of DataAccess class and manages the lifetime of the object.

Problems in the above example classes:

1. CustomerBusinessLogic and DataAccess classes are tightly coupled classes. So, changes in the DataAccess class will lead to changes in the CustomerBusinessLogic class. For example, if we add, remove or rename any method in the DataAccess class then we need to change the CustomerBusinessLogic class accordingly.
2. Suppose the customer data comes from different databases or web services and, in the future, we may need to create different classes, so this will lead to changes in the CustomerBusinessLogic class.
3. The CustomerBusinessLogic class creates an object of the DataAccess class using the **new** keyword. There may be multiple classes which use the DataAccess class and create its objects. So, if you change the name of the class, then you need to find all the places in your source code where you created objects of DataAccess and make the changes throughout the code. This is repetitive code for creating objects of the same class and maintaining their dependencies.
4. Because the CustomerBusinessLogic class creates an object of the concrete DataAccess class, it cannot be tested independently (TDD). The DataAccess class cannot be replaced with a mock class.

To solve all of the above problems and get a loosely coupled design, we can use the IoC and DIP principles together. Remember, IoC is a principle, not a pattern. It just gives high-level design guidelines but does not give implementation details. You are free to implement the IoC principle the way you want.

The following pattern (but not limited) implements the IoC principle.

[](https://www.tutorialsteacher.com/Content/images/ioc/ioc-patterns.png)

Let's use the *Factory* pattern to implement IoC in the above example, as the first step towards attaining loosely coupled classes.

First, create a simple Factory class which returns an object of the DataAccess class as shown below.

Example: DataAccess Factory

public class DataAccessFactory

{

public static DataAccess GetDataAccessObj()

{

return new DataAccess();

}

}

Now, use this DataAccessFactory class in the CustomerBusinessLogic class to get an object of DataAccess class.

Example: Use Factory Class to Retrieve Object

public class CustomerBusinessLogic

{

public CustomerBusinessLogic()

{

}

public string GetCustomerName(int id)

{

DataAccess \_dataAccess = DataAccessFactory.GetDataAccessObj();

return \_dataAccess.GetCustomerName(id);

}

}

As you can see, the CustomerBusinessLogic class uses the DataAccessFactory.GetCustomerDataAccessObj() method to get an object of the DataAccess class instead of creating it using the *new* keyword. Thus, we have inverted the control of creating an object of a dependent class from the CustomerBusinessLogic class to the DataAccessFactory class.

This is a simple implementation of IoC and the first step towards achieving fully loose coupled design. As mentioned in the previous chapter, we will not achieve complete loosely coupled classes by only using IoC. Along with IoC, we also need to use DIP, Strategy pattern, and DI (Dependency Injection).

1. What is Dependency Injection?

Ans **Dependency Injection** is passing dependency to other **objects** or **framework**( dependency injector).

Dependency injection makes testing easier. The injection can be done through **constructor**.

SomeClass() has its constructor as following:

public SomeClass() {

myObject = Factory.getObject();

}

**Problem**: If myObject involves complex tasks such as disk access or network access, it is **hard** to do unit test on SomeClass(). Programmers have to mock myObject and might **intercept** the factory call.

**Alternative solution**:

* Passing myObject in as an argument to the constructor

public SomeClass (MyClass myObject) {

this.myObject = myObject;

}

myObject can be passed directly which makes testing easier.

* One common alternative is defining a **do-nothing constructor**. Dependency injection can be done through setters. (h/t @MikeVella). It is harder to isolate components in unit testing without dependency injection.

1. Can you give few examples of Dependency Injection?

Ans Java Dependency injection seems hard to grasp with theory, so I would take a simple example and then we will see how to use dependency injection pattern to achieve loose coupling and extendability in the application.

Let’s say we have an application where we consume EmailService to send emails. Normally we would implement this like below.

package com.journaldev.java.legacy;

public class EmailService {

public void sendEmail(String message, String receiver){

//logic to send email

System.out.println("Email sent to "+receiver+ " with Message="+message);

}

}

EmailService class holds the logic to send an email message to the recipient email address. Our application code will be like below.

package com.journaldev.java.legacy;

public class MyApplication {

private EmailService email = new EmailService();

public void processMessages(String msg, String rec){

//do some msg validation, manipulation logic etc

this.email.sendEmail(msg, rec);

}

}

Our client code that will use MyApplication class to send email messages will be like below.

package com.journaldev.java.legacy;

public class MyLegacyTest {

public static void main(String[] args) {

MyApplication app = new MyApplication();

app.processMessages("Hi Pankaj", "pankaj@abc.com");

}

}

At first look, there seems nothing wrong with the above implementation. But above code logic has certain limitations.

* MyApplication class is responsible to initialize the email service and then use it. This leads to hard-coded dependency. If we want to switch to some other advanced email service in the future, it will require code changes in MyApplication class. This makes our application hard to extend and if email service is used in multiple classes then that would be even harder.
* If we want to extend our application to provide an additional messaging feature, such as SMS or Facebook message then we would need to write another application for that. This will involve code changes in application classes and in client classes too.
* Testing the application will be very difficult since our application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

One can argue that we can remove the email service instance creation from MyApplication class by having a constructor that requires email service as an argument.

package com.journaldev.java.legacy;

public class MyApplication {

private EmailService email = null;

public MyApplication(EmailService svc){

this.email=svc;

}

public void processMessages(String msg, String rec){

//do some msg validation, manipulation logic etc

this.email.sendEmail(msg, rec);

}

}

But in this case, we are asking client applications or test classes to initializing the email service that is not a good design decision.

Now let’s see how we can apply java dependency injection pattern to solve all the problems with the above implementation. Dependency Injection in java requires at least the following:

1. Service components should be designed with base class or interface. It’s better to prefer interfaces or abstract classes that would define contract for the services.
2. Consumer classes should be written in terms of service interface.
3. Injector classes that will initialize the services and then the consumer classes.

### Java Dependency Injection – Service Components

For our case, we can have MessageService that will declare the contract for service implementations.

package com.journaldev.java.dependencyinjection.service;

public interface MessageService {

void sendMessage(String msg, String rec);

}

Now let’s say we have Email and SMS services that implement the above interfaces.

package com.journaldev.java.dependencyinjection.service;

public class EmailServiceImpl implements MessageService {

@Override

public void sendMessage(String msg, String rec) {

//logic to send email

System.out.println("Email sent to "+rec+ " with Message="+msg);

}

}

package com.journaldev.java.dependencyinjection.service;

public class SMSServiceImpl implements MessageService {

@Override

public void sendMessage(String msg, String rec) {

//logic to send SMS

System.out.println("SMS sent to "+rec+ " with Message="+msg);

}

}

Our dependency injection java services are ready and now we can write our consumer class.

### Java Dependency Injection – Service Consumer

We are not required to have base interfaces for consumer classes but I will have a Consumer interface declaring contract for consumer classes.

package com.journaldev.java.dependencyinjection.consumer;

public interface Consumer {

void processMessages(String msg, String rec);

}

My consumer class implementation is like below.

package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

private MessageService service;

public MyDIApplication(MessageService svc){

this.service=svc;

}

@Override

public void processMessages(String msg, String rec){

//do some msg validation, manipulation logic etc

this.service.sendMessage(msg, rec);

}

}

Notice that our application class is just using the service. It does not initialize the service that leads to better “separation of concerns“. Also use of service interface allows us to easily test the application by mocking the MessageService and bind the services at runtime rather than compile time.

Now we are ready to write **java dependency injector classes** that will initialize the service and also consumer classes.

### Java Dependency Injection – Injectors Classes

Let’s have an interface MessageServiceInjector with method declaration that returns the Consumer class.

package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

public interface MessageServiceInjector {

public Consumer getConsumer();

}

Now for every service, we will have to create injector classes like below.

package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;

import com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

@Override

public Consumer getConsumer() {

return new MyDIApplication(new EmailServiceImpl());

}

}

package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;

import com.journaldev.java.dependencyinjection.service.SMSServiceImpl;

public class SMSServiceInjector implements MessageServiceInjector {

@Override

public Consumer getConsumer() {

return new MyDIApplication(new SMSServiceImpl());

}

}

Now let’s see how our client applications will use the application with a simple program.

package com.journaldev.java.dependencyinjection.test;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

import com.journaldev.java.dependencyinjection.injector.EmailServiceInjector;

import com.journaldev.java.dependencyinjection.injector.MessageServiceInjector;

import com.journaldev.java.dependencyinjection.injector.SMSServiceInjector;

public class MyMessageDITest {

public static void main(String[] args) {

String msg = "Hi Pankaj";

String email = "pankaj@abc.com";

String phone = "4088888888";

MessageServiceInjector injector = null;

Consumer app = null;

//Send email

injector = new EmailServiceInjector();

app = injector.getConsumer();

app.processMessages(msg, email);

//Send SMS

injector = new SMSServiceInjector();

app = injector.getConsumer();

app.processMessages(msg, phone);

}

}

As you can see that our application classes are responsible only for using the service. Service classes are created in injectors. Also if we have to further extend our application to allow facebook messaging, we will have to write Service classes and injector classes only.

So dependency injection implementation solved the problem with hard-coded dependency and helped us in making our application flexible and easy to extend. Now let’s see how easily we can test our application class by mocking the injector and service classes.

### Java Dependency Injection – JUnit Test Case with Mock Injector and Service

package com.journaldev.java.dependencyinjection.test;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;

import com.journaldev.java.dependencyinjection.injector.MessageServiceInjector;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplicationJUnitTest {

private MessageServiceInjector injector;

@Before

public void setUp(){

//mock the injector with anonymous class

injector = new MessageServiceInjector() {

@Override

public Consumer getConsumer() {

//mock the message service

return new MyDIApplication(new MessageService() {

@Override

public void sendMessage(String msg, String rec) {

System.out.println("Mock Message Service implementation");

}

});

}

};

}

@Test

public void test() {

Consumer consumer = injector.getConsumer();

consumer.processMessages("Hi Pankaj", "pankaj@abc.com");

}

@After

public void tear(){

injector = null;

}

}

As you can see that I am using [anonymous classes](https://www.journaldev.com/996/java-inner-class) to mock the injector and service classes and I can easily test my application methods. I am using JUnit 4 for the above test class, so make sure it’s in your project build path if you are running above test class.

We have used constructors to inject the dependencies in the application classes, another way is to use a setter method to **inject dependencies** in application classes. For setter method dependency injection, our application class will be implemented like below.

package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

private MessageService service;

public MyDIApplication(){}

//setter dependency injection

public void setService(MessageService service) {

this.service = service;

}

@Override

public void processMessages(String msg, String rec){

//do some msg validation, manipulation logic etc

this.service.sendMessage(msg, rec);

}

}

package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;

import com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

@Override

public Consumer getConsumer() {

MyDIApplication app = new MyDIApplication();

app.setService(new EmailServiceImpl());

return app;

}

}

One of the best example of setter dependency injection is [Struts2 Servlet API Aware interfaces](https://www.journaldev.com/2203/get-servlet-session-request-response-context-attributes-struts-2-action).

Whether to use Constructor based dependency injection or setter based is a design decision and depends on your requirements. For example, if my application can’t work at all without the service class then I would prefer constructor based DI or else I would go for setter method based DI to use it only when it’s really needed.

**Dependency Injection in Java** is a way to achieve **Inversion of control** (**IoC**) in our application by moving objects binding from compile time to runtime. We can achieve IoC through [Factory Pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java), [Template Method Design Pattern](https://www.journaldev.com/1763/template-method-design-pattern-in-java), [Strategy Pattern](https://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial) and Service Locator pattern too.

[**Spring Dependency Injection**](https://www.journaldev.com/2410/spring-dependency-injection), [**Google Guice**](https://www.journaldev.com/2403/google-guice-dependency-injection-example-tutorial) and **Java EE CDI** frameworks facilitate the process of dependency injection through use of [Java Reflection API](https://www.journaldev.com/1789/java-reflection-example-tutorial) and [java annotations](https://www.journaldev.com/721/java-annotations). All we need is to annotate the field, constructor or setter method and configure them in configuration xml files or classes.

### Benefits of Java Dependency Injection

Some of the benefits of using Dependency Injection in Java are:

* Separation of Concerns
* Boilerplate Code reduction in application classes because all work to initialize dependencies is handled by the injector component
* Configurable components makes application easily extendable
* Unit testing is easy with mock objects

### Disadvantages of Java Dependency Injection

Java Dependency injection has some disadvantages too:

* If overused, it can lead to maintenance issues because the effect of changes are known at runtime.
* Dependency injection in java hides the service class dependencies that can lead to runtime errors that would have been caught at compile time.

1. What is Auto Wiring?

In Spring framework, you can wire beans automatically with auto-wiring feature. To enable it, just define the “**autowire**” attribute in <bean>.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

Copy

In Spring, 5 Auto-wiring modes are supported.

* no – Default, no auto wiring, set it manually via “ref” attribute
* byName – Auto wiring by property name. If the name of a bean is same as the name of other bean property, auto wire it.
* byType – Auto wiring by property data type. If data type of a bean is compatible with the data type of other bean property, auto wire it.
* constructor – byType mode in constructor argument.
* autodetect – If a default constructor is found, use “autowired by constructor”; Otherwise, use “autowire by type”.

## **Examples**

A Customer and Person object for auto wiring demonstration.

package com.mkyong.common;

public class Customer

{

private Person person;

public Customer(Person person) {

this.person = person;

}

public void setPerson(Person person) {

this.person = person;

}

//...

}

Copy

package com.mkyong.common;

public class Person

{

//...

}

Copy

## **1. Auto-Wiring ‘no’**

This is the default mode, you need to wire your bean via ‘ref’ attribute.

<bean id="customer" class="com.mkyong.common.Customer">

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

</bean>

<bean id="person" class="com.mkyong.common.Person" />

Copy

## **2. Auto-Wiring ‘byName’**

Auto-wire a bean by property name. In this case, since the name of “person” bean is same with the name of the “customer” bean’s property (“person”), so, Spring will auto wired it via setter method – “setPerson(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="person" class="com.mkyong.common.Person" />

Copy

See full example – [Spring Autowiring by Name](http://www.mkyong.com/spring/spring-autowiring-by-name/).

## **3. Auto-Wiring ‘byType’**

Auto-wire a bean by property data type. In this case, since the data type of “person” bean is same as the data type of the “customer” bean’s property (Person object), so, Spring will auto wired it via setter method – “setPerson(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byType" />

<bean id="person" class="com.mkyong.common.Person" />

Copy

See full example – [Spring Autowiring by Type](http://www.mkyong.com/spring/spring-autowiring-by-type/).

## **4. Auto-Wiring ‘constructor’**

Auto-wire a bean by property data type in constructor argument. In this case, since the data type of “person” bean is same as the constructor argument data type in “customer” bean’s property (Person object), so, Spring auto wired it via constructor method – “public Customer(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="constructor" />

<bean id="person" class="com.mkyong.common.Person" />

Copy

See full example – [Spring Autowiring by Constructor](http://www.mkyong.com/spring/spring-autowiring-by-constructor/).

## **5. Auto-Wiring ‘autodetect’**

If a default constructor is found, uses “constructor”; Otherwise, uses “byType”. In this case, since there is a default constructor in “Customer” class, so, Spring auto wired it via constructor method – “public Customer(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="autodetect" />

<bean id="person" class="com.mkyong.common.Person" />

Copy

See full example – [Spring Autowiring by AutoDetect](http://www.mkyong.com/spring/spring-autowiring-by-autodetect/).

**Note**  
It’s always good to combine both ‘auto-wire’ and ‘dependency-check’ together, to make sure the property is always auto-wire successfully.

<bean id="customer" class="com.mkyong.common.Customer"

autowire="autodetect" dependency-check="objects />

<bean id="person" class="com.mkyong.common.Person" />

Copy

## **Conclusion**

In my view, Spring ‘auto-wiring’ make development faster with great costs – it added complexity for the entire bean configuration file, and you don’t even know which bean will auto wired in which bean.

In practice, i rather wire it manually, it is always clean and work perfectly, or better uses [@Autowired annotation](http://www.mkyong.com/spring/spring-auto-wiring-beans-with-autowired-annotation/), which is more flexible and recommended.

1. What are the important roles of an IOC Container?

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

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

There are two types of IoC containers. They are:

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

### 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.

### Using BeanFactory

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:

1. Resource resource=**new** ClassPathResource("applicationContext.xml");
2. 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.

#### Using ApplicationContext

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

1. ApplicationContext context =
2. **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.

1. What are Bean Factory and Application Context?

### Ans ****The BeanFactory Interface****

This is the root interface for accessing the Spring container. To access the Spring container, we will be using Spring's dependency injection functionality using this BeanFactory interface and its sub-interfaces.

Features:

* Bean instantiation/wiring

Usually, the implementations use lazy loading, which means that beans are only instantiating when we directly calling them through the getBean() method.

The most used API that implements the **BeanFactory** is the **XmlBeanFactory**.

Here is an example of how to get a bean through the BeanFactory:

package com.zoltanraffai;

import org.springframework.core.io.ClassPathResource;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.xml.XmlBeanFactory;

public class HelloWorldApp{

public static void main(String[] args) {

XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("beans.xml"));

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");

obj.getMessage();

}

}

### ****The ApplicationContext Interface****

The **ApplicationContext**is the central interface within a Spring application that is used for providing configuration information to the application.

It implements the BeanFactory interface. Hence, the ApplicationContext includes all functionality of the BeanFactory and much more! Its main function is to support the creation of big business applications.

Features:

* Bean instantiation/wiring
* Automatic BeanPostProcessor registration
* Automatic BeanFactoryPostProcessor registration
* Convenient MessageSource access (for i18n)
* ApplicationEvent publication

It uses eager loading, so every bean instantiate after the ApplicationContext is started up.

Here is an example of the **ApplicationContext** usage:

package com.zoltanraffai;

import org.springframework.core.io.ClassPathResource;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.xml.XmlBeanFactory;

public class HelloWorldApp{

public static void main(String[] args) {

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

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

obj.getMessage();

}

}

## Conclusion

The **ApplicationContext**includes all the functionality of the **BeanFactory.** It is generally recommended to use the former. There are some limited situations, such as in mobile applications, where memory consumption might be critical. In those scenarios, it would be justifiable to use the more lightweight **BeanFactory**. However, in most enterprise applications, the **ApplicationContext** is what you will want to use.

1. Can you compare Bean Factory with Application Context?

Ans 1) BeanFactory doesn't provide support for internationalization i.e. i18n but ApplicationContext provides support for it.

2) Another difference between BeanFactory vs ApplicationContext is ability to publish event to beans that are registered as listener.

3) One of the popular implementation of BeanFactory interface is XMLBeanFactory while one of the popular implementation of ApplicationContext interface is ClassPathXmlApplicationContext. On [Java web application](http://javarevisited.blogspot.sg/2012/08/what-is-jsessionid-in-j2ee-web.html) we use WebApplicationContext  which extends ApplicationContext interface and adds getServletContext method.

4) If you are using auto wiring and using BeanFactory than you need to register AutoWiredBeanPostProcessor using API which you can configure in XML if you are using  ApplicationContext. In summary BeanFactory is OK for testing and non [production](http://javarevisited.blogspot.in/2011/09/how-to-write-production-quality-code.html) use but ApplicationContext is more feature rich container implementation and should be favored over BeanFactory

These were some worth noting difference between BeanFactory and ApplicationContext in Spring framework. In most practical cases you will be using ApplicationContext but knowing about BeanFactory is important to understand fundamental concept of spring framework. I mostly use XML configuration file and ClassPathXmlApplicationContext to quickly run any Spring based Java program from [Eclipse](http://javarevisited.blogspot.sg/2012/10/eclipse-shortcut-to-remove-all-unused-imports-java.html) by using following snippet of code :

**public** **static** **void** main(**String** args[]){  
    ApplicationContext ctx =**new** ClassPathXmlApplicationContext("beans.xml");  
    Hello hello =(Hello) ctx.getBean("hello");  
    hello.sayHello("John");  
}

here beans.xml is your spring configuration file and “hello” is a bean defined in that spring configuration file. Here we have used ClassPathXmlApplicationContext  which is an implementation of ApplicationContext interface in Spring.

1. How do you create an application context with Spring?

Ans The Application Context is Spring's advanced container. Similar to BeanFactory, it can load bean definitions, wire beans together, and dispense beans upon request. Additionally, it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by *org.springframework.context.ApplicationContext* interface.

The *ApplicationContext* includes all functionality of the *BeanFactory*, It is generally recommended over BeanFactory. BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications.

The most commonly used ApplicationContext implementations are −

* **FileSystemXmlApplicationContext** − This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
* **ClassPathXmlApplicationContext** − This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look like bean configuration XML file in CLASSPATH.
* **WebXmlApplicationContext** − This container loads the XML file with definitions of all beans from within a web application.

We already have seen an example on ClassPathXmlApplicationContext container in *Spring Hello World Example*, and we will talk more about XmlWebApplicationContext in a separate chapter when we will discuss web-based Spring applications. So let us see one example on FileSystemXmlApplicationContext.

## **Example**

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *HelloWorld* and *MainApp* under the *com.tutorialspoint* package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **HelloWorld.java** file −

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

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

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new FileSystemXmlApplicationContext

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

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

obj.getMessage();

}

}

Following two important points should be noted about the main program −

* The first step is to create factory object where we used framework API**FileSystemXmlApplicationContext** to create the factory bean after loading the bean configuration file from the given path. The**FileSystemXmlApplicationContext()** API takes care of creating and initializing all the objects ie. beans mentioned in the XML bean configuration file.
* The second step is used to get the required bean using **getBean()** method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

Following is the content of the bean configuration file **Beans.xml**

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

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>

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

Your Message : Hello World!

1. How does Spring know where to search for Components or Beans?

If you understand Component Scan, you understand Spring.

Spring is a dependency injection framework. It is all about beans and wiring in dependencies.

The first step of defining Spring Beans is by adding the right annotation — @Component or @Service or @Repository.

However, Spring does not know about the bean unless it knows where to search for it.

**This part of “telling Spring where to search” is called a Component Scan.**

You define the packages that have to be scanned.

Once you define a Component Scan for a package, Spring would search the package and all its sub packages for components/beans.

### Defining a Component Scan

* If you are using Spring Boot, check the configuration in Approach 1.
* If you are doing a JSP/Servlet or a Spring MVC application without using Spring Boot, use Approach 2.

## Approach 1: Component Scan in a Spring Boot Project

* If your other package hierarchies are below your main app with the @SpringBootApplication annotation, you’re covered by the implicit Component Scan.
* If there are beans/components in other packages that are not sub-packages of the main package, you should manually add them as @ComponentScan

Consider the class below:

package com.in28minutes.springboot.basics.springbootin10steps;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication

public class SpringbootIn10StepsApplication {

public static void main(String[] args) {

ApplicationContext applicationContext =

SpringApplication.run(SpringbootIn10StepsApplication.class, args);

for (String name: applicationContext.getBeanDefinitionNames()) {

System.out.println(name);

}

}

}

@SpringBootApplication is defined in the SpringbootIn10StepsApplication class which is in the package com.in28minutes.springboot.basics.springbootin10steps

@SpringBootApplication defines an automatic Component Scan on the package com.in28minutes.springboot.basics.springbootin10steps.

You are fine if all your components are defined in the above package or a sub-package of it.

However, let’s say one of the components is defined in package com.in28minutes.springboot.somethingelse

In this case, you would need to add the new package into Component Scan.

You have two options:

* Define @ComponentScan(“com.in28minutes.springboot”)
  + This would scan the entire parent tree of com.in28minutes.springboot.
* Or define two specific Component Scans by using an array.
  + @ComponentScan({“com.in28minutes.springboot.basics.springbootin10steps”,”com.in28minutes.springboot.somethingelse”})

Option 1:

@ComponentScan(“com.in28minutes.springboot”)

@SpringBootApplication

public class SpringbootIn10StepsApplication {

Option 2:

@ComponentScan({"com.in28minutes.springboot.basics.springbootin10steps","com.in28minutes.springboot.somethingelse"})

@SpringBootApplication

public class SpringbootIn10StepsApplication {

## Approach 2: Non-Spring Boot Project

In a non-Spring Boot Project, we would typically define the component scan explicitly in an XML application context or a Java Application Context.

Option 1:

@ComponentScan(“com.in28minutes)

@Configuration

public class SpringConfiguration {

Option 2:

@ComponentScan({"com.in28minutes.package1","com.in28minutes.package2"})

@Configuration

public class SpringConfiguration {

XML application context:

<context:component-scan base-package="com.in28minutes" />

Specific multiple packages:

<context:component-scan base-package="com.in28minutes.package1, com.in28minutes.package2" />

### URL Not Working

The server starts up fine, but:

* My URL is not working
* My login URL is not working
* My todo URL is not working ``` WARNING: No mapping found for HTTP request with URI [/spring-mvc/login] in DispatcherServlet with name ‘dispatcher’ WARNING: No mapping found for HTTP request with URI [/login] in DispatcherServlet with name ‘dispatcher’ WARNING: No mapping found for HTTP request with URI [/list-todos] in DispatcherServlet with name ‘dispatcher’

#### No qualifying bean of type found

No qualifying bean of type [com.in28minutes.springboot.jpa.UserRepository] found for dependency [com.in28minutes.springboot.jpa.UserRepository]: expected at least 1 bean which qualifies as autowire candidate for this dependency. Dependency annotations: {@org.springframework.beans.factory.annotation.Autowired(required=true)}

```

**Same root cause for both above problems — the component is not being picked up.**

There are three possible things you would need to look at:

* You have not added the right annotation — @Controller, @Repository, or @Controller
* You have not added a Component Scan.
* The package of your component is not defined in Component Scan.

You have two options:

1. Add the annotation or component scan
2. Move the component to a package already under Component Scan

## @Component vs. @ComponentScan

@Component and @ComponentScan are for different purposes.

* @Component indicates that a class might be a candidate for creating a bean. It's like putting a hand up.
* @ComponentScan is searching packages for Components. Trying to find out who all put their hands up.

1. What is a Component Scan?

When developing Spring Boot applications, you need to tell the Spring Framework where to look for Spring components. Using component scan is one method of asking Spring to detect Spring-managed components. Spring needs the information to locate and register all the Spring components with the application context when the application starts.

Spring can auto scan, detect, and instantiate components from pre-defined project packages. Spring can auto scan all classes annotated with the stereotype annotations @Component, @Controller, @Service, and @Repository

In this post, I will discuss how Spring component scanning works.

## Sample Application

Let's create a simple Spring Boot application to understand how component scanning works in Spring.

We will start by writing few components.

DemoBeanA.java:

package guru.springframework.blog.componentscan.example.demopackageA;

import org.springframework.stereotype.Component;

@Component("demoBeanA")

public class DemoBeanA {

}

DemoBeanB1.java:

package guru.springframework.blog.componentscan.example.demopackageB;

import org.springframework.stereotype.Component;

@Component("demoBeanB1")

public class DemoBeanB1 {

}

DemoBeanB2.java:

package guru.springframework.blog.componentscan.example.demopackageB;

import org.springframework.stereotype.Component;

@Component("demoBeanB2")

public class DemoBeanB2 extends DemoBeanB1{

}

DemoBeanB3.java:

package guru.springframework.blog.componentscan.example.demopackageB;

import org.springframework.stereotype.Component;

@Component("demoBeanB3")

public class DemoBeanB3 extends DemoBeanB2{

}

DemoBeanC.java:

package guru.springframework.blog.componentscan.example.demopackageC;

import org.springframework.stereotype.Component;

@Component("demoBeanC")

public class DemoBeanC {

}

DemoBeanD.java:

package guru.springframework.blog.componentscan.example.demopackageD;

import org.springframework.stereotype.Component;

@Component("demoBeanD")

public class DemoBeanD {

}

## The @SpringBootApplication Annotation

Spring needs to know which packages to scan for annotated components in order to add them to the IoC container. In a Spring Boot project, we typically set the main application class with the @SpringBootApplication annotation. Under the hood, @SpringBootApplication is a composition of the @Configuration, @ComponentScan, and @EnableAutoConfiguration annotations. With this default setting, Spring Boot will auto scan for components in the current package (containing the @SpringBoot main class) and its sub packages.

To know more about these annotations, go through my [Spring Framework Annotations](https://springframework.guru/spring-framework-annotations/) post.

Note: It is recommended that you locate your main application class in a root package above the component classes of the application.

The code to create the main class and access components i:

BlogPostsApplication.java

package guru.springframework.blog;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class BlogPostsApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(BlogPostsApplication.class,args);

System.out.println("Contains A "+context.

containsBeanDefinition("demoBeanA"));

System.out.println("Contains B2 " + context.

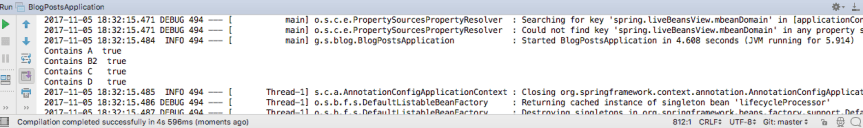
containsBeanDefinition("demoBeanB2"));

System.out.println("Contains C " + context.

containsBeanDefinition("demoBeanC"));

}

}

The output of running the main class is:  
[](https://i0.wp.com/springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplication.java_.png)

As you can see, all the classes in the subpackages of the main class BlogPostsApplication are auto scanned by Spring.

## @ComponentScan: Identifying Base Packages

The @ComponentScan annotation is used with the @Configuration annotation to tell Spring the packages to scan for annotated components. @ComponentScan also used to specify base packages and base package classes using thebasePackageClasses or basePackages attributes of @ComponentScan.

The basePackageClasses attribute is a type-safe alternative to basePackages. When you specify basePackageClasses, Spring will scan the package (and subpackages) of the classes you specify.

A Java class annotated with @ComponentScan with the basePackageClassesattribute is:

BlogPostsApplicationWithComponentScan.java:

package guru.springframework.blog;

import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB1;

import org.springframework.boot.SpringApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = {

"guru.springframework.blog.componentscan.example.demopackageA",

"guru.springframework.blog.componentscan.example.demopackageD",

"guru.springframework.blog.componentscan.example.demopackageE"

},

basePackageClasses = DemoBeanB1.class)

public class BlogPostsApplicationWithComponentScan {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationWithComponentScan.class, args);

System.out.println("Contains A " + context.containsBeanDefinition("demoBeanA"));

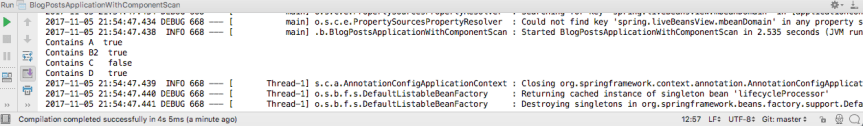
System.out.println("Contains B2 " + context.containsBeanDefinition("demoBeanB2"));

System.out.println("Contains C " + context.containsBeanDefinition("demoBeanC"));

System.out.println("Contains D " + context.containsBeanDefinition("demoBeanD"));

}

}

The output of running the main class is:  
[](https://i2.wp.com/springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationWithComponentScan.java_.png)

The @ComponentScan annotation uses the basePackages attribute to specify three packages (and subpackages) that will be scanned by Spring. The annotation also uses the basePackageClasses attribute to declare the DemoBeanB1 class whose package Spring Boot should scan.

As demoBeanC is in a different package, Spring did not find it during component scanning.

## Component Scanning Filters

You can configure component scanning by using different type filters that Spring provides.

By using filters, you can further narrow the set of candidate components from everything in basePackages to everything in the base packages that matches the given filter or filters.

Filters can be of two types: include and exclude filters. As their names suggest, include filters specify which types are eligible for component scanning, while exclude filters specify which types are not.

You can use the include and/or exclude filters with or without the default filter. To disable the default filter, set the useDefaultFilters element of the @ComponentScan annotation to false.

The code to disable the default filter is:

BlogPostsApplicationDisablingDefaultFilters.java:

package guru.springframework.blog;

import org.springframework.boot.SpringApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(value = "guru.springframework.blog.componentscan.example.demopackageA",

useDefaultFilters = false)

public class BlogPostsApplicationDisablingDefaultFilters {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

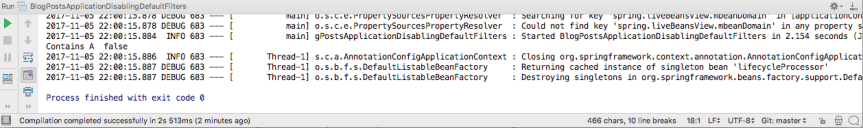
run(BlogPostsApplicationDisablingDefaultFilters.class,args);

System.out.println("Contains A " + context.containsBean("demoBeanA"));

}

}

In the preceding code, the value member defines the specific guru.springframework.blog.componentscan.example.demopackageA package to scan, while the useDefaultFilters member disables the default filter.

The output of running the main class is:  
[](https://i0.wp.com/springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationDisablingDefaultFilters.java_.png)

As you can see, the class DemoBeanA in the package demopackageA is unavailable when the useDefaultFilters element of the @ComponentScan annotation is set to false.

### Component Scanning Filter Types

Spring provides the FilterType enumeration for the type filters that may be used in conjunction with @ComponentScan.

The available FilterType values are:

* FilterType.ANNOTATION: Include or exclude those classes with a stereotype annotation
* FilterType.ASPECTJ: Include or exclude classes using an AspectJ type pattern expression
* FilterType.ASSIGNABLE\_TYPE: Include or exclude classes that extend or implement this class or interface
* FilterType.REGEX: Include or exclude classes using a regular expression
* FilterType.CUSTOM: Include or exclude classes using a custom implementation of the org.springframework.core.type.TypeFilter interface

### Include Filters

With include filters, you can include certain classes to be scanned by Spring. To include assignable type, use the includeFilters element of the @ComponentScan annotation with FilterType.ASSIGNABLE\_TYPE. Using this filter, you can instruct Spring to scan for classes that extends or implements the class or interface you specify.

The code to use the includeFilters element of @ComponentScan is:

BlogPostsApplicationIncludeFilter.java:

package guru.springframework.blog;

import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;

import org.springframework.boot.SpringApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.FilterType;

@Configuration

@ComponentScan(basePackages = {"guru.springframework.blog.componentscan.example.demopackageA",

"guru.springframework.blog.componentscan.example.demopackageB"},

includeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, value = DemoBeanB2.class),

useDefaultFilters = false)

public class BlogPostsApplicationIncludeFilter {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationIncludeFilter.class,args);

System.out.println("Contains A " + context.containsBean("demoBeanA"));

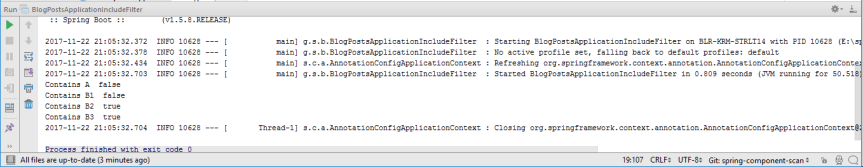
System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));

System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));

System.out.println("Contains B3 " + context.containsBean("demoBeanB3"));

}

}

The output of running the main class is:  
[](https://i1.wp.com/springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationIncludeFilter.java_.png)

As shown in the preceding figure, Spring detected and used the demoBean3 component that extends demoBean2.

### Include Filters Using Regex

You can use regular expressions to filter out components to be scanned by Spring. Use the includeFiltersnested annotation @ComponentScan.Filter type FilterType.REGEXto set a pattern.

The code to use an exclude filter based on regular expression is:

BlogPostsApplicationFilterTypeRegex.java:

package guru.springframework.blog;

import org.springframework.boot.SpringApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.FilterType;

@Configuration

@ComponentScan(useDefaultFilters = false, includeFilters = @ComponentScan.Filter

(type = FilterType.REGEX, pattern = ".\*[A2]"))

public class BlogPostsApplicationFilterTypeRegex {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationFilterTypeRegex.class,args);

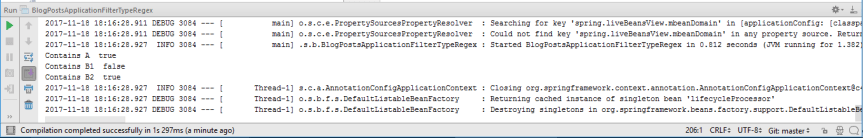
System.out.println("Contains A " + context.containsBean("demoBeanA"));

System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));

System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));

}

}

The output of the following code snippet is:  
[](https://i2.wp.com/springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationFilterTypeRegex.java_.png)  
As shown in the preceding figure, the classes whose names end with A or 2 are detected by Spring.

### Exclude Filters

The @ComponentScan annotation enables you to exclude those classes that you do not want to scan.

The code to use an exclude filter is:

BlogPostsApplicationExcludeFilter.java:

package guru.springframework.blog;

import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB1;

import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;

import org.springframework.boot.SpringApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.FilterType;

@Configuration

@ComponentScan(basePackageClasses = {DemoBeanB1.class},

excludeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE,

value = DemoBeanB2.class))

public class BlogPostsApplicationExcludeFilter {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.

run(BlogPostsApplicationExcludeFilter.class,args);

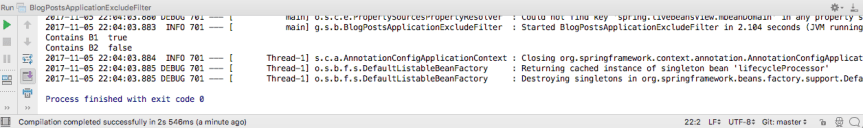
System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));

System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));

}

}

In this code, the nested annotation @ComponentScan.Filter is used to specify the filter type as FilterType.ASSIGNABLE\_TYPE and the base class that should be excluded from scanning.

The output is:  
[](https://i2.wp.com/springframework.guru/wp-content/uploads/2017/11/Output-of-using-the-FilterType.ASSIGNABLE_TYPE-type.png)

As you can see, the class DemoBeanB2 has been excluded from being scanned.

## Summary

When using Spring Boot, most of the time, the default auto scanning will work for your project. You only need to ensure that your @SpringBoot main class is at the base package of your package hierarchy. Spring Boot will automatically perform a component scan in the package of the Spring Boot main class and below.

One related annotation that I didn’t mention in this post is that @EntityScan is more about JPA entity scanning rather than component scanning. The @EntityScan annotation, unlike @ComponentScan, does not create beans. It only identifies which classes should be used by a specific persistence context.

1. How do you define a component scan in XML and Java Configurations?

Normally you declare all the beans or components in XML bean configuration file, so that Spring container can detect and register your beans or components. Actually, Spring is able to auto scan, detect and instantiate your beans from pre-defined project package, no more tedious beans declaration in in XML file.

Following is a simple Spring project, including a customer service and dao layer. Let’s explore the different between declare components manually and auto components scanning in Spring.

## **1. Declares Components Manually**

See a normal way to declare a bean in Spring.

Normal bean.

package com.mkyong.customer.dao;

public class CustomerDAO

{

@Override

public String toString() {

return "Hello , This is CustomerDAO";

}

}

Copy

DAO layer.

package com.mkyong.customer.services;

import com.mkyong.customer.dao.CustomerDAO;

public class CustomerService

{

CustomerDAO customerDAO;

public void setCustomerDAO(CustomerDAO customerDAO) {

this.customerDAO = customerDAO;

}

@Override

public String toString() {

return "CustomerService [customerDAO=" + customerDAO + "]";

}

}

Copy

Bean configuration file (Spring-Customer.xml), a normal bean configuration in Spring.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="customerService" class="com.mkyong.customer.services.CustomerService">

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

</bean>

<bean id="customerDAO" class="com.mkyong.customer.dao.CustomerDAO" />

</beans>

Copy

Run it

package com.mkyong.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App

{

public static void main( String[] args )

{

ApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"Spring-Customer.xml"});

CustomerService cust = (CustomerService)context.getBean("customerService");

System.out.println(cust);

}

}

Copy

output

CustomerService [customerDAO=Hello , This is CustomerDAO]

Copy

## **2. Auto Components Scanning**

Now, enable Spring auto component scanning features.

Annotate with **@Component** to indicate this is class is an auto scan component.

package com.mkyong.customer.dao;

import org.springframework.stereotype.Component;

@Component

public class CustomerDAO

{

@Override

public String toString() {

return "Hello , This is CustomerDAO";

}

}

Copy

DAO layer, add **@Component** to indicate this is an auto scan component also.

package com.mkyong.customer.services;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import com.mkyong.customer.dao.CustomerDAO;

@Component

public class CustomerService

{

@Autowired

CustomerDAO customerDAO;

@Override

public String toString() {

return "CustomerService [customerDAO=" + customerDAO + "]";

}

}

Copy

Put this “context:component” in bean configuration file, it means, enable auto scanning feature in Spring. The **base-package** is indicate where are your components stored, Spring will scan this folder and find out the bean (annotated with @Component) and register it in Spring container.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="com.mkyong.customer" />

</beans>

Copy

Run it

package com.mkyong.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App

{

public static void main( String[] args )

{

ApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"Spring-AutoScan.xml"});

CustomerService cust = (CustomerService)context.getBean("customerService");

System.out.println(cust);

}

}

Copy

output

CustomerService [customerDAO=Hello , This is CustomerDAO]

Copy

This is how auto components scanning works in Spring.

## **Custom auto scan component name**

By default, Spring will lower case the first character of the component – from ‘CustomerService’ to ‘customerService’. And you can retrieve this component with name ‘customerService’.

CustomerService cust = (CustomerService)context.getBean("customerService");

Copy

To create a custom name for component, you can put custom name like this :

@Service("AAA")

public class CustomerService

...

Copy

Now, you can retrieve it with this name ‘AAA’.

CustomerService cust = (CustomerService)context.getBean("AAA");

Copy

## **Auto Components Scan Annotation Types**

In Spring 2.5, there are 4 types of auto components scan annotation types

* @Component – Indicates a auto scan component.
* @Repository – Indicates DAO component in the persistence layer.
* @Service – Indicates a Service component in the business layer.
* @Controller – Indicates a controller component in the presentation layer.

So, which one to use? It’s really doesn’t matter. Let see the source code of @Repository,@Service or @Controller.

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Repository {

String value() default "";

}

Copy

You will noticed that all @Repository,@Service or @Controller are annotated with @Component. So, can we use just @Component for all the components for auto scanning? Yes, you can, and Spring will auto scan all your components with @Component annotated.

It’s working fine, but not a good practice, for readability, you should always declare @Repository,@Service or @Controller for a specified layer to make your code more easier to read, as following :

DAO layer

package com.mkyong.customer.dao;

import org.springframework.stereotype.Repository;

@Repository

public class CustomerDAO

{

@Override

public String toString() {

return "Hello , This is CustomerDAO";

}

}

Copy

Service layer

package com.mkyong.customer.services;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.mkyong.customer.dao.CustomerDAO;

@Service

public class CustomerService

{

@Autowired

CustomerDAO customerDAO;

@Override

public String toString() {

return "CustomerService [customerDAO=" + customerDAO + "]";

}

}

1. How is it done with Spring Boot?

### Ans Defining a Component Scan

* If you are using Spring Boot, check the configuration in Approach 1.
* If you are doing a JSP/Servlet or a Spring MVC application without using Spring Boot, use Approach 2.

## Approach 1: Component Scan in a Spring Boot Project

* If your other package hierarchies are below your main app with the @SpringBootApplication annotation, you’re covered by the implicit Component Scan.
* If there are beans/components in other packages that are not sub-packages of the main package, you should manually add them as @ComponentScan

Consider the class below:

package com.in28minutes.springboot.basics.springbootin10steps;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication

public class SpringbootIn10StepsApplication {

public static void main(String[] args) {

ApplicationContext applicationContext =

SpringApplication.run(SpringbootIn10StepsApplication.class, args);

for (String name: applicationContext.getBeanDefinitionNames()) {

System.out.println(name);

}

}

}

@SpringBootApplication is defined in the SpringbootIn10StepsApplication class which is in the package com.in28minutes.springboot.basics.springbootin10steps

@SpringBootApplication defines an automatic Component Scan on the package com.in28minutes.springboot.basics.springbootin10steps.

You are fine if all your components are defined in the above package or a sub-package of it.

However, let’s say one of the components is defined in package com.in28minutes.springboot.somethingelse

In this case, you would need to add the new package into Component Scan.

You have two options:

* Define @ComponentScan(“com.in28minutes.springboot”)
  + This would scan the entire parent tree of com.in28minutes.springboot.
* Or define two specific Component Scans by using an array.
  + @ComponentScan({“com.in28minutes.springboot.basics.springbootin10steps”,”com.in28minutes.springboot.somethingelse”})

Option 1:

@ComponentScan(“com.in28minutes.springboot”)

@SpringBootApplication

public class SpringbootIn10StepsApplication {

Option 2:

@ComponentScan({"com.in28minutes.springboot.basics.springbootin10steps","com.in28minutes.springboot.somethingelse"})

@SpringBootApplication

public class SpringbootIn10StepsApplication {

## Approach 2: Non-Spring Boot Project

In a non-Spring Boot Project, we would typically define the component scan explicitly in an XML application context or a Java Application Context.

Option 1:

@ComponentScan(“com.in28minutes)

@Configuration

public class SpringConfiguration {

Option 2:

@ComponentScan({"com.in28minutes.package1","com.in28minutes.package2"})

@Configuration

public class SpringConfiguration {

XML application context:

<context:component-scan base-package="com.in28minutes" />

Specific multiple packages:

<context:component-scan base-package="com.in28minutes.package1, com.in28minutes.package2" />

### URL Not Working

The server starts up fine, but:

* My URL is not working
* My login URL is not working
* My todo URL is not working ``` WARNING: No mapping found for HTTP request with URI [/spring-mvc/login] in DispatcherServlet with name ‘dispatcher’ WARNING: No mapping found for HTTP request with URI [/login] in DispatcherServlet with name ‘dispatcher’ WARNING: No mapping found for HTTP request with URI [/list-todos] in DispatcherServlet with name ‘dispatcher’

#### No qualifying bean of type found

No qualifying bean of type [com.in28minutes.springboot.jpa.UserRepository] found for dependency [com.in28minutes.springboot.jpa.UserRepository]: expected at least 1 bean which qualifies as autowire candidate for this dependency. Dependency annotations: {@org.springframework.beans.factory.annotation.Autowired(required=true)}

```

**Same root cause for both above problems — the component is not being picked up.**

There are three possible things you would need to look at:

* You have not added the right annotation — @Controller, @Repository, or @Controller
* You have not added a Component Scan.
* The package of your component is not defined in Component Scan.

You have two options:

1. Add the annotation or component scan
2. Move the component to a package already under Component Scan

## @Component vs. @ComponentScan

@Component and @ComponentScan are for different purposes.

* @Component indicates that a class might be a candidate for creating a bean. It's like putting a hand up.
* @ComponentScan is searching packages for Components. Trying to find out who all put their hands up.

1. What does @Component signify?

## Ans Spring Component

In layman terms, a Component is responsible for some operations. [Spring framework](https://www.journaldev.com/16922/spring-framework) provides three other specific annotations to be used when marking a class as Component.

1. Service: Denotes that the class provides some services. Our utility classes can be marked as Service classes.
2. Repository: This annotation indicates that the class deals with CRUD operations, usually it’s used with [DAO](https://www.journaldev.com/16813/dao-design-pattern) implementations that deal with database tables.
3. Controller: Mostly used with [web applications](https://www.journaldev.com/14476/spring-mvc-example) or [REST web services](https://www.journaldev.com/2552/spring-rest-example-tutorial-spring-restful-web-services) to specify that the class is a front controller and responsible to handle user request and return appropriate response.

Note that all these four annotations are in package org.springframework.stereotype and part of spring-context jar.

Most of the time our component classes will fall under one of its three specialized annotations, so you may not use @Component annotation a lot.

## Spring Component Example

Let’s create a very simple Spring maven application to showcase the use of Spring Component annotation and how Spring autodetects it with annotation-based configuration and classpath scanning.

Create a maven project and add following spring core dependency.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.0.6.RELEASE</version>

</dependency>

That’s all we need to get the spring framework core features.

Let’s create a simple component class and mark it with @Component annotation.

package com.journaldev.spring;

import org.springframework.stereotype.Component;

@Component

public class MathComponent {

public int add(int x, int y) {

return x + y;

}

}

Now we can create an annotation based spring context and get the MathComponent bean from it.

package com.journaldev.spring;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringMainClass {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

context.scan("com.journaldev.spring");

context.refresh();

MathComponent ms = context.getBean(MathComponent.class);

int result = ms.add(1, 2);

System.out.println("Addition of 1 and 2 = " + result);

context.close();

}

}

Just run the above class as normal java application and you should get following output in the console.

Jun 05, 2018 12:49:26 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@ff5b51f: startup date [Tue Jun 05 12:49:26 IST 2018]; root of context hierarchy

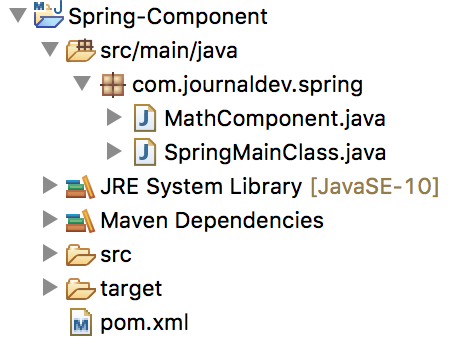
Addition of 1 and 2 = 3

Jun 05, 2018 12:49:26 PM org.springframework.context.support.AbstractApplicationContext doClose

INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@ff5b51f: startup date [Tue Jun 05 12:49:26 IST 2018]; root of context hierarchy

Did you realized the power of Spring, we didn’t have to do anything to inject our component to spring context.

Below image shows the directory structure of our Spring Component example project.



We can also specify the component name and then get it from spring context using the same name.

@Component("mc")

public class MathComponent {

}

MathComponent ms = (MathComponent) context.getBean("mc");

Although I have used @Component annotation with MathComponent, it’s actually a service class and we should use @Service annotation. The result will still be the same.

1. What does @Autowired signify?

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.

### Example

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *TextEditor*, *SpellChecker* and *MainApp* under the *com.tutorialspoint* package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **TextEditor.java** file −

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker( ) {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java**:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

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

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

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

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

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xmlns:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

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

</bean>

<!-- Definition for spellChecker bean -->

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

</bean>

</beans>

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

Inside SpellChecker constructor.

Inside checkSpelling.

## **@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 your **TextEditor.java** file will become as follows −

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

@Autowired

private SpellChecker spellChecker;

public TextEditor() {

System.out.println("Inside TextEditor constructor." );

}

public SpellChecker getSpellChecker( ){

return spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

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

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

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xmlns:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for textEditor bean -->

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

</bean>

<!-- Definition for spellChecker bean -->

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

</bean>

</beans>

Once you are done with the above two changes in source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message −

Inside TextEditor constructor.

Inside SpellChecker constructor.

Inside checkSpelling.

## **@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.

Here is the content of **TextEditor.java** file −

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

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

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

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xmlns:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

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

</bean>

<!-- Definition for spellChecker bean -->

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

</bean>

</beans>

Once you are done with the above two changes in source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

## **@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.

The following example will work even if you do not pass any value for age property but still it will demand for name property. You can try this example yourself because this is similar to @Required annotation example except that only **Student.java** file has been changed.

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class Student {

private Integer age;

private String name;

@Autowired(required=false)

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Autowired

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

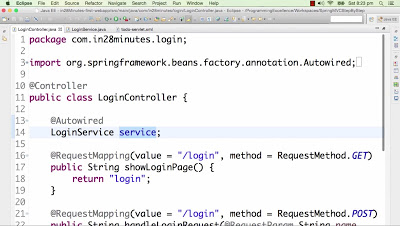
}

}

1. What’s the difference Between @Controller, @Component, @Repository, and @Service Annotations in Spring?

Ans Before you learn the difference between @Component, @Service, @Controller, and @Repository annotations in [Spring framework](https://javarevisited.blogspot.com/2018/06/top-6-spring-framework-online-courses-Java-programmers.html), it's important to understand the role of @Component annotation in Spring. During initial release of Spring, all beans are used to be declared in an XML file. For a large project, this quickly becomes a massive task and Spring guys recognize the problem rather quickly. In later versions, they provide annotation-based dependency injection and Java-based configuration. From Spring 2.5 annotation-based [dependency injection](http://javarevisited.blogspot.sg/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) was introduced, which automatically scans and register classes as Spring bean which is annotated using @Component annotation.  
  
This means you don't declare that bean using the <bean> tag and inject the dependency, it will be done automatically by Spring. This functionality was enabled and disabled using <context:component-scan> tag.  
  
Now that you know what does @Component annotation does let's see what does @Service, @Controller, and @Repository annotation do.  
  
They are nothing but the specialized form of @Component annotation for certain situations. Instead of using @Component on a controller class in Spring MVC, we use @Controller, which is more readable and appropriate.  
  
By using that annotation we do two things, first we declare that this class is a Spring bean and should be created and maintained by [Spring ApplicationContext](http://javarevisited.blogspot.sg/2012/11/difference-between-beanfactory-vs-applicationcontext-spring-framework.html), but also we indicate that its a controller in MVC setup. This latter property is used by web-specific tools and functionalities.

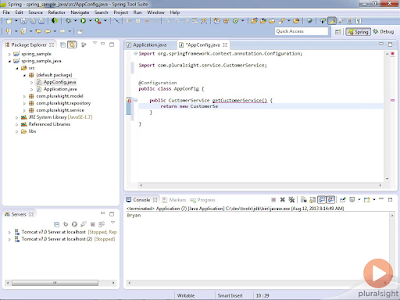
For example, DispatcherServlet will look for @RequestMapping on classes which are annotated using @Controller but not with @Component.  
  
This means @Component and @Controller are same with respect to bean creation and dependency injection but later is a specialized form of former. Even if you replace @Controller annotation with @Compoenent, Spring can automatically detect and register the controller class but it may not work as you expect with respect to request mapping. You can further see, [**Spring Master Class**](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fspring-tutorial-for-beginners%2F) course on Udemy for more details about these annotations.

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fspring-tutorial-for-beginners%2F)

Same is true for @Service and @Repository annotation, they are a specialization of @Component in service and persistence layer. A Spring bean in the service layer should be annotated using @Service instead of @Component annotation and a spring bean in the persistence layer should be annotated with @Repository annotation.  
  
By using a specialized annotation we hit two birds with one stone. First, they are treated as Spring bean and second you can put special behavior required by that layer.  
  
For example, @Repository's not only helping in annotation based configure but also catch Platform specific exceptions and re-throw them as one of Spring’s unified [unchecked exception](https://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html).  
  
Though for that you also need to declare org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor as Spring bean in your application context.  
  
This bean post processor adds an advisor to any bean that’s annotated with @Repository so that any platform-specific exceptions are caught and then rethrown as one of Spring’s unchecked data access exceptions. You can also see [**Spring Framework 5: Beginner to Guru**](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fspring-framework-5-beginner-to-guru%2F) on Udemy for more details.  
  
This is also one of the frequently asked Spring Interview Question and a popular concept from the Spring certification perspective. You will find a couple of questions based on these annotations and their usage in Spring professional certification exam too.

## How does Component Scanning work in Spring?

From Spring 2.0, Spring provides <context:component-scan> and annotation-driven dependency injection to automatically detect and register Spring bean instead of specifying them in the XML file.  
  
But, it only scans @Component and does not look for @Controller, @Service and @Repository in general. They are scanned because they themselves are annotated with @Component.  
  
Just take a look at @Controller, @Service, and @Repository annotation definitions:  
  
@Component  
public @interface Service {  
….  
}  
  
  
@Component  
public @interface Repository {  
….  
}  
  
  
@Component  
public @interface Controller {  
…  
}  
  
Thus, it’s not wrong to say that @Controller, @Service, and @Repository are special types of @Component annotation. <context:component-scan> picks them up and registers their following classes as beans, just as if they were annotated with @Component.  
  
They are scanned because they themselves are annotated with @Component annotation. If you define your own custom annotation and annotate it with @Component, then it will also get scanned with <context:component-scan>.  
  
If you want to learn more about dependency injection, auto-wiring and different types of configuration in Spring e.g. XML based, annotation-based and Java configuration in Spring, I suggest you take the [**Spring Fundamentals course**](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fspring-fundamentals) on Pluralsight. A free trial is also available.

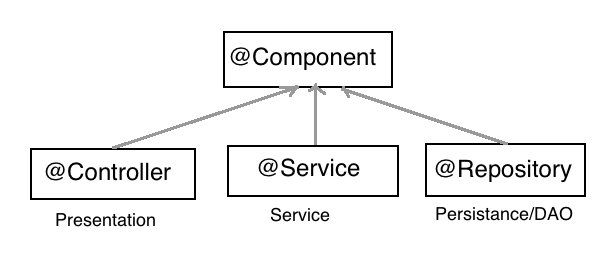
[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fspring-mvc4-introduction)

### Summary

Here is a nice summary of what does @Component, @Service, @Controller, and @Repository annotation do in Spring Framework:

1. @Component is a generic stereotype for any Spring-managed component or bean.
2. @Repository is a stereotype for the persistence layer.
3. @Service is a stereotype for the service layer.
4. @Controller is a stereotype for the presentation layer (spring-MVC).

And here is the nice diagram to explain the hierarchy of all these annotations in Spring Framework:

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fspring-interview-questions-preparation-course%2F)

That's all about the **difference between @Component, @Controller, @Service, and @Repository in Spring Framework**. As I said, all of them are used to auto-detect Spring beans when context scanning is enabled and essentially provide the same functionality with respect to dependency injection.  
  
Their only difference comes in their purpose i.e. @Controller is used in Spring MVC to define controller, which are first Spring bean and then controller. Similarly, @Service is used to annotated classes which hold business logic in the Service layer and @Repository is used in the Data Access layer.  
  
You can read more about component scanning and how Spring framework automatically detects bean in [Spring fundamentals course](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fspring-fundamentals) by Bryna Hassen on Pluralsight. You can get it free for 10 days as well.  
  
In short, you should use the most appropriate annotation based upon which layer that particular class belongs.

1. What is the default scope of a bean?

Ans From the [spring specs](http://static.springsource.org/spring/docs/3.0.0.M3/reference/html/ch04s04.html), there are five types of bean scopes supported :

**1. singleton(default\*)**

Scopes a single bean definition to a single object instance per Spring IoC container.

**2. prototype**

Scopes a single bean definition to any number of object instances.

**3. request**

Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.

**4. session**

Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

**5. global session**

Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

1. Are Spring beans thread safe?

The short answer is: no, it isn’t.

And you probably already know why.

It’s because of the long life cycle of singleton beans. Those beans may be reused over and over again in many HTTP requests coming from different users.

If you don’t use [@Lazy](http://dolszewski.com/spring/spring-lazy-annotation-use-cases/)**, the framework creates a singleton bean at the application startup** and makes sure that the same instance is autowired and reused in all other dependent beans. As long the container lives, the singleton beans live as well.

But the framework doesn’t control how the singleton is used. If two different threads execute a method of the singleton at the same time, you’re not guaranteed that both calls will be synchronized and run in the sequence.

In other words, it’s **your responsibility** to ensure your code runs safely in the multithreaded environment. Spring won’t do that for you.

1. What are the other scopes available?

**1. singleton(default\*)**

Scopes a single bean definition to a single object instance per Spring IoC container.

**2. prototype**

Scopes a single bean definition to any number of object instances.

**3. request**

Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.

**4. session**

Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

**5. global session**

Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

1. How is Spring’s singleton bean different from Gang of Four Singleton Pattern?

The Gang of Four defines Singleton as having only one instance per ClassLoader whereas Spring singleton is defined as one instance of bean definition per container.

1. What are the different types of dependency injections?

### Introduction

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), **dependency injection** is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a [service](https://en.wikipedia.org/wiki/Service_(systems_architecture))).

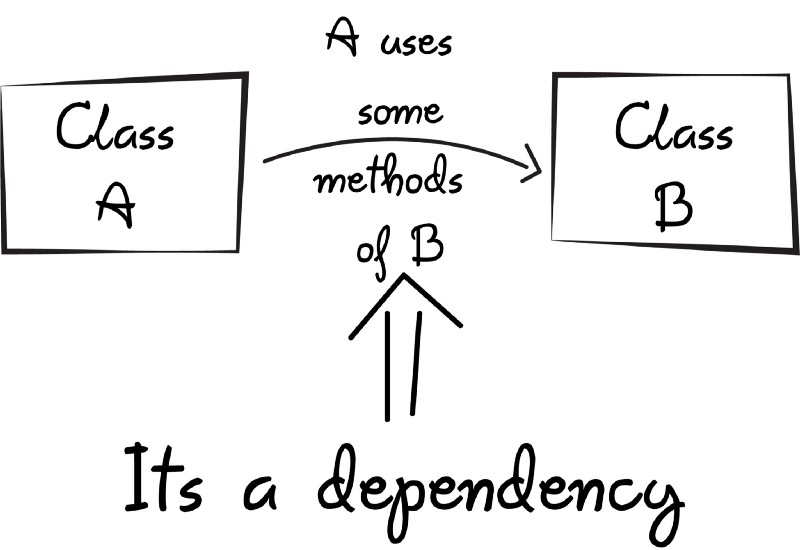
That’s the Wikipedia definition but still, but it’s not particularly easy to understand. So let’s understand it better.

Before understanding what it means in programming, let’s first see what it means in general as it will help us understand the concept better.

Dependency or dependent means relying on something for support. Like if I say we are relying too much on mobile phones than it means we are dependent on them.

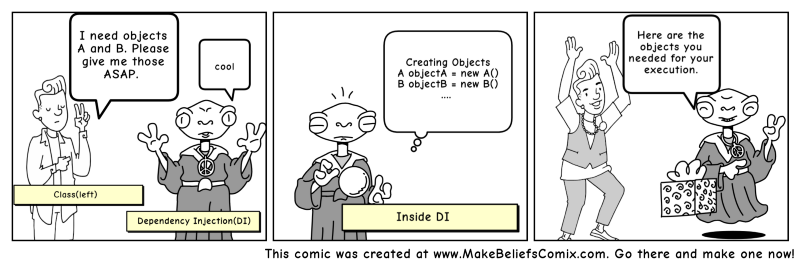
So before getting to [**dependency injections**](https://en.wikipedia.org/wiki/Dependency_injection), first let’s understand what a dependency in programming means.

When class A uses some functionality of class B, then its said that class A has a dependency of class B.

Showing dependencies between classes

In Java, before we can use methods of other classes, we first need to create the object of that class (i.e. class A needs to create an instance of class B).

**So, transferring the task of creating the object to someone else and directly using the dependency is called dependency injection.**

What if code could speak?

### Why should I use dependency injection?

Let’s say we have a car class which contains various objects such as wheels, engine, etc.

Here the car class is responsible for creating all the dependency objects. Now, what if we decide to ditch **MRFWheels**in the future and want to use **Yokohama** Wheels?

We will need to recreate the car object with a new Yokohama dependency. But when using dependency injection (DI), we can change the Wheels at runtime (because dependencies can be injected at runtime rather than at compile time).

You can think of DI as the middleman in our code who does all the work of creating the preferred wheels object and providing it to the Car class.

It makes our Car class independent from creating the objects of Wheels, Battery, etc.

#### There are basically three types of dependency injection:

1. **constructor injection:** the dependencies are provided through a class constructor.
2. **setter injection:** the client exposes a setter method that the injector uses to inject the dependency.
3. **interface injection:** the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a [setter method](https://en.wikipedia.org/wiki/Setter_method) that accepts the dependency.

**So now its the dependency injection’s responsibility to:**

1. Create the objects
2. Know which classes require those objects
3. And provide them all those objects

If there is any change in objects, then DI looks into it and it should not concern the class using those objects. This way if the objects change in the future, then its DI’s responsibility to provide the appropriate objects to the class.

#### Inversion of control —the concept behind DI

This states that a class should not configure its dependencies statically but should be configured by some other class from outside.

It is the fifth principle of **S.O.L.I.D** — thefive basic principles of object-oriented programming and design by [**Uncle Bob**](https://en.wikipedia.org/wiki/Robert_C._Martin) — which states that a class should depend on abstraction and not upon concretions (in simple terms, hard-coded).

According to the principles, a class should concentrate on fulfilling its responsibilities and not on creating objects that it requires to fulfill those responsibilities. And that’s where **dependency injection** comes into play: it provides the class with the required objects.

Note: If you want to learn about ***SOLID***principles by Uncle Bob then you can head to this [*link*](https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#toc-single-responsibility-principle).

#### Benefits of using DI

1. Helps in Unit testing.
2. Boiler plate code is reduced, as initializing of dependencies is done by the injector component.
3. Extending the application becomes easier.
4. Helps to enable loose coupling, which is important in application programming.

#### Disadvantages of DI

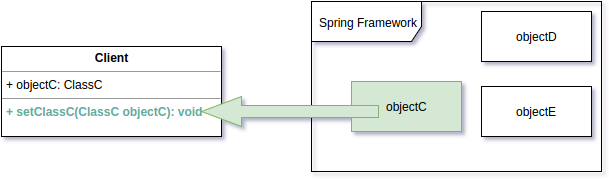
1. It’s a bit complex to learn, and if overused can lead to management issues and other problems.
2. Many compile time errors are pushed to run-time.
3. Dependency injection frameworks are implemented with reflection or dynamic programming. This can hinder use of IDE automation, such as “find references”, “show call hierarchy” and safe refactoring.

You can implement dependency injection on your own (Pure Vanilla) or use third-party libraries or frameworks.

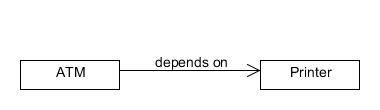
#### ****Libraries and Frameworks that implement DI****

* [Spring](https://www.tutorialspoint.com/spring/spring_dependency_injection.htm)(Java)
* [Google Guice](https://github.com/google/guice) (Java)
* [Dagger](http://square.github.io/dagger/)(Java and Android)
* [Castle Windsor](https://github.com/castleproject/Windsor) (.NET)
* [Unity](https://www.microsoft.com/en-us/download/details.aspx?id=39944)(.NET)

1. What is setter injection?

Setter Injection in Spring is a type of dependency injection in which the framework injects the dependent objects into the client using a setter method. The container first calls the no argument constructor and then calls the setters. The setter based injection can work even If some dependencies have been injected using the constructor.  
[Github – Source Code for this example](https://github.com/MithilShah/Spring-Examples/tree/master/spring_setter_injection)  
[](http://d18fvftsc9acw8.cloudfront.net/wp-content/uploads/2016/09/SetterInjection.png)

## Example of Setter Injection in Spring

In typical software development, classes collaborate with each other to achieve a desired functionality. e.g. In an ATM (Automated Teller Machine) system, the ATM class and Printer class can collaborate with each other to print the balance information for a bank account.  
  
  


|  |  |
| --- | --- |
| 1  2  3  4 | public class ATM {                private Printer printer;              ... |

The dependencies need to be ‘resolved’ before the desired functionality is can be achieved. e.g. By ‘resolved’ we mean that an instance of Printer class needs to be created and associated with the ‘printer’ member in ATM class. When dependency resolution is not performed by the class itself but is left to be done by an external agent (e.g. Spring Framework) it is called dependency injection. So in our example, Spring will create an instance of the Printer class and associate the instance with the ‘printer’ member in ATM class. However as the ‘printer’ member in ATM class is private, the ATM class needs to expose its dependency to Spring for it to inject the Printer instance into the ATM class. e.g. If the ATM class exposes its dependency on Printer class as a setter method so that Spring can inject the Printer object then this is called as Setter injection (see line 12 below).

|  |  |
| --- | --- |
|  | package com.studytrails.tutorials.springsetterinjection; |
|  |  |
|  | public class ATM { |
|  |  |
|  | private Printer printer; |
|  |  |
|  | public Printer getPrinter() { |
|  | return printer; |
|  | } |
|  |  |
|  | public void setPrinter(Printer printer) { |
|  | this.printer = printer; |
|  | } |
|  |  |
|  | public void printBalanceInformation(String accountNumber) { |
|  | getPrinter().printBalanceInformation(accountNumber); |
|  |  |
|  | } |
|  |  |
|  | } |

[**view raw**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece/raw/0e8f85bb225fa3b40834ae4fc77ed7aec7f46f05/spring_setter_injection-s-c-s-t-s-ATM.java)[**spring\_setter\_injection-s-c-s-t-s-ATM.java**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece#file-spring_setter_injection-s-c-s-t-s-atm-java) hosted with  by [**GitHub**](https://github.com/)

Now, Spring can create an instance of Printer class and inject it via the setter method defined in ATM class.

### *Construction injection or Setter Injection ?*

Setter method injection is a very common way to resolve dependencies within the Spring framework. Hence, although constructor injection is possible using Spring framework, setter method injection is the most favored way to resolve dependencies using Spring framework.

## Sample Program for Setter Injection in spring

### *Program Overview*

This program uses spring framework’s setter injection mechanism to inject and instance of Printer class into ATM.  
Here’s the pom file.

|  |  |
| --- | --- |
|  | <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" |
|  | xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"> |
|  | <modelVersion>4.0.0</modelVersion> |
|  | <groupId>spring\_setter\_injection</groupId> |
|  | <artifactId>spring\_setter\_injection</artifactId> |
|  | <version>0.0.1-SNAPSHOT</version> |
|  | <name>spring\_setter\_injection</name> |
|  | <description>spring\_setter\_injection</description> |
|  | <build> |
|  | <sourceDirectory>src</sourceDirectory> |
|  | <plugins> |
|  | <plugin> |
|  | <artifactId>maven-compiler-plugin</artifactId> |
|  | <version>3.5.1</version> |
|  | <configuration> |
|  | <source>1.8</source> |
|  | <target>1.8</target> |
|  | </configuration> |
|  | </plugin> |
|  | </plugins> |
|  | </build> |
|  | <dependencies> |
|  | <!-- https://mvnrepository.com/artifact/org.springframework/spring-context --> |
|  | <dependency> |
|  | <groupId>org.springframework</groupId> |
|  | <artifactId>spring-context</artifactId> |
|  | <version>4.3.4.RELEASE</version> |
|  | </dependency> |
|  |  |
|  | </dependencies> |
|  | </project> |

[**view raw**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece/raw/0e8f85bb225fa3b40834ae4fc77ed7aec7f46f05/spring_setter_injection-pom.xml)[**spring\_setter\_injection-pom.xml**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece#file-spring_setter_injection-pom-xml) hosted with  by [**GitHub**](https://github.com/)

### *Source Code for Setter injection in spring example*

Create a Printer class with a method printBalanceInformation()

|  |  |
| --- | --- |
|  | package com.studytrails.tutorials.springsetterinjection; |
|  |  |
|  | public class Printer { |
|  |  |
|  | public void printBalanceInformation(String accountNumber) { |
|  | System.out.println("The printer has printed the balance information for the account number " + accountNumber); |
|  |  |
|  | } |
|  |  |
|  | } |

[**view raw**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece/raw/0e8f85bb225fa3b40834ae4fc77ed7aec7f46f05/spring_setter_injection-s-c-s-t-s-Printer.java)[**spring\_setter\_injection-s-c-s-t-s-Printer.java**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece#file-spring_setter_injection-s-c-s-t-s-printer-java) hosted with  by [**GitHub**](https://github.com/)

Create the ATM class with a dependency on Printer class. The ATM class delegates the call to print the balance information to the Printer class (see line 16 in the ATM class above). Now before Spring framework can perform setter injection, it needs to know about the ATM and Printer classes. This is achieved by declaring the ATM and Printer classes as Spring beans in spring-config.xml (see line 10 and line 14 below).Note that setter injection is achieved by passing a reference of the ‘printer’ class to ATM class bean definition(see line 11 below)

|  |  |
| --- | --- |
|  | <?xml version="1.0" encoding="UTF-8"?> |
|  | <beans xmlns="http://www.springframework.org/schema/beans" |
|  | xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop" |
|  | xsi:schemaLocation=" |
|  | http://www.springframework.org/schema/beans |
|  | http://www.springframework.org/schema/beans/spring-beans-3.0.xsd |
|  | http://www.springframework.org/schema/aop |
|  | http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"> |
|  |  |
|  | <bean id="atm" class="com.studytrails.tutorials.springsetterinjection.ATM"> |
|  | <property name="printer" ref="printer" /> |
|  | </bean> |
|  |  |
|  | <bean id="printer" |
|  | class="com.studytrails.tutorials.springsetterinjection.Printer"> |
|  | </bean> |
|  |  |
|  | </beans> |

[**view raw**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece/raw/0e8f85bb225fa3b40834ae4fc77ed7aec7f46f05/spring_setter_injection-s-spring-config.xml)[**spring\_setter\_injection-s-spring-config.xml**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece#file-spring_setter_injection-s-spring-config-xml) hosted with  by [**GitHub**](https://github.com/)

Finally, we need a java program to test our setter dependency injection setup. This is done by TestSetterDependencyInjection.java (see source code below). We need to tell Spring framework to use the ‘spring-config.xml’ to load our beans(see line 11 below). We get the reference to ATM class through Spring using the bean name ‘atm’ (see line 12 below). In this step, Spring will resolve the dependency on Printer class by injecting it using setter injection. We call the printBalanceInformation() on ATM class (see line 14 below) with some accountNumber (see line 13 below).

|  |  |
| --- | --- |
|  | package com.studytrails.tutorials.springsetterinjection; |
|  |  |
|  | import org.springframework.context.ApplicationContext; |
|  | import org.springframework.context.support.ClassPathXmlApplicationContext; |
|  |  |
|  | public class TestSetterDependencyInjection { |
|  |  |
|  | public static void main(String[] args) { |
|  | System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\* BEGINNING PROGRAM \*\*\*\*\*\*\*\*\*\*\*\*\*\*"); |
|  |  |
|  | ApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml"); |
|  | ATM atm = (ATM)context.getBean("atm"); |
|  | String accountNumber = "AC5645786"; |
|  | atm.printBalanceInformation(accountNumber); |
|  |  |
|  |  |
|  | System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\* ENDING PROGRAM \*\*\*\*\*\*\*\*\*\*\*\*\*\*"); |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece/raw/0e8f85bb225fa3b40834ae4fc77ed7aec7f46f05/spring_setter_injection-s-c-s-t-s-TestSetterDependencyInjection.java)[**spring\_setter\_injection-s-c-s-t-s-TestSetterDependencyInjection.java**](https://gist.github.com/MithilShah/12a5c24c7bc7ff751ddd14010bfd6ece#file-spring_setter_injection-s-c-s-t-s-testsetterdependencyinjection-java) hosted with  by [**GitHub**](https://github.com/)

This completes our tutorial on Setter Injection in Spring. In the next tutorial, we will look at a less used form of injection – spring inner bean injection

1. What is constructor injection?

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

## **Example**

The following example shows a class *TextEditor* that can only be dependency-injected with constructor injection.

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *TextEditor*, *SpellChecker* and *MainApp* under the *com.tutorialspoint* package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **TextEditor.java** file −

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java**

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

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

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** which has configuration for the constructor-based injection −

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

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

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

<constructor-arg ref = "spellChecker"/>

</bean>

<!-- Definition for spellChecker bean -->

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

</beans>

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

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

## **Constructor arguments resolution**

There may be an ambiguity while passing arguments to the constructor, in case there are more than one parameters. To resolve this ambiguity, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor. Consider the following class −

package x.y;

public class Foo {

public Foo(Bar bar, Baz baz) {

// ...

}

}

The following configuration works fine −

<beans>

<bean id = "foo" class = "x.y.Foo">

<constructor-arg ref = "bar"/>

<constructor-arg ref = "baz"/>

</bean>

<bean id = "bar" class = "x.y.Bar"/>

<bean id = "baz" class = "x.y.Baz"/>

</beans>

Let us check one more case where we pass different types to the constructor. Consider the following class −

package x.y;

public class Foo {

public Foo(int year, String name) {

// ...

}

}

The container can also use type matching with simple types, if you explicitly specify the type of the constructor argument using the type attribute. For example −

<beans>

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

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

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

</bean>

</beans>

Finally, the best way to pass constructor arguments, use the index attribute to specify explicitly the index of constructor arguments. Here, the index is 0 based. For example −

<beans>

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

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

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

</bean>

</beans>

A final note, in case you are passing a reference to an object, you need to use **ref** attribute of <constructor-arg> tag and if you are passing a value directly then you should use **value** attribute as shown above.

1. How do you choose between setter and constructor injections?

Ans 25 Spring supports two types of dependency Injection, using setter method e.g. setXXX() where XXX is a dependency or via a constructor argument. The first way of dependency injection is known as **setter injection** while later is known as **constructor injection**. Both approaches of Injecting dependency on Spring bean has there pros and cons, which we will see in this Spring framework article. *The difference between Setter Injection and Constructor Injection in Spring* is also a popular [Spring framework interview question](http://javarevisited.blogspot.sg/2011/09/spring-interview-questions-answers-j2ee.html).Some time interviewer also asks as When do you use Setter Injection over Constructor injection in Spring or simply benefits of using setter vs constructor injection in Spring framework. Points discussed in this article not only help you to understand Setter vs Constructor Injection but also Spring's dependency Injection process.   
  
By the way, if you are new in Spring framework and learning it, you may want to take a look at my list of [5 good books to learn Spring framework](http://javarevisited.blogspot.sg/2013/03/5-good-books-to-learn-spring-framework-mvc-java-programmer.html). That will certainly help in  your learning process. Since Spring is now a must have skill for Java programmers

, it worth putting time and effort to learn this powerful framework

## **Difference between Setter and Constructor Injection in Spring framework**

[](https://2.bp.blogspot.com/-jlIT7z2RuQg/T9whph8c-FI/AAAAAAAAAYw/TrAvT017-1c/s1600/spring_thumbnail.PNG)As I said earlier Spring supports both setter and constructor Injection which are two standard way of injecting dependency on beans managed by IOC constructor. Spring framework doesn't support Interface Injection on which dependency is injected by implementing a particular interface. In this section we will see a couple of difference between setter and constructor Injection, which will help you decide when to use setter Injection over constructor Injection in Spring and vice-versa.

1) The fundamental difference between setter and constructor injection, as their name implies is How dependency is injected.  Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container. On the other hand constructor injection uses [constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) to inject dependency on any Spring-managed bean.

2) Because of using setter method, setter Injection in more readable than constructor injection in Spring configuration

 file usually applicationContext.xml . Since setter method has name e.g. setReporotService() by reading Spring XML config file you know which dependency you are setting. While in constructor injection, since it uses an index to inject the dependency, it's not as readable as setter injection and you need to refer either Java documentation or code to find which index corresponds to which property.

3) Another difference between setter vs constructor injection in Spring and one of the drawback of  setter injection is that it does not ensures [dependency Injection](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html). You can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On other hand constructor Injection does not allow you to construct object, until your dependencies are ready.

4) One more drawback of setter Injection is Security. By using setter injection, you can [override](http://javarevisited.blogspot.in/2011/12/method-overloading-vs-method-overriding.html) certain dependency which is not possible which is not possible with constructor injection because every time you call the constructor, a new object is gets created.  
  
  
5) One of our reader Murali Mohan Reddy pointed out one more difference between Setter and Constructor injection in Spring, where later can help if there is a circular dependency between two object A and B.

If Object A and B are dependent each other i.e A is depends ob B and vice-versa. Spring throws ObjectCurrentlyInCreationException while creating objects of A and B bcz A object cannot be created until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection. Objects constructed before setter methods invoked.

See comment section for more inputs from other readers.

## When to use Setter Injection over Constructor Injection in Spring

Setter Injection has upper hand over Constructor Injection in terms of readability. Since for configuring Spring we use [XML files](http://javarevisited.blogspot.in/2011/12/parse-xml-file-in-java-example-tutorial.html), readability is much bigger concern. Also drawback of setter Injection around ensuring mandatory dependency injected or not can be handled by configuring Spring to check dependency using "dependency-check" attribute of  tag or tag. Another worth noting point to remember while comparing Setter Injection vs Constructor Injection is that, once number of dependency crossed a threshold e.g. 5 or 6 its handy manageable to passing dependency via constructor. Setter Injection is preferred choice when number of dependency to be injected is lot more than normal, if some of those arguments is optional than using [Builder design pattern](http://javarevisited.blogspot.in/2012/06/builder-design-pattern-in-java-example.html) is also a good option.

In Summary, both Setter Injection and Constructor Injection has there own advantage and disadvantage. The good thing about Spring is that it doesn't restrict you to use either Setter Injection or Constructor Injection and you are free to use both of them in one Spring configuration file. Use Setter injection when a number of dependencies are more or you need readability. Use Constructor Injection when Object must be created with all of its dependency.

1. What are the different options available to create Application Contexts for Spring?

Ans The **Spring IoC container** is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application.

Spring provides following two distinct types of containers.

1. BeanFactory container
2. ApplicationContext container

## 1. Spring BeanFactory container

A **BeanFactory** is essentially nothing more than the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. The BeanFactory enables you to read bean definitions and access them using the bean factory. When using just the BeanFactory you would create one and read in some bean definitions in the XML format as follows:

|  |
| --- |
| How to Create XmlBeanFactory |
| InputStream is = new FileInputStream("beans.xml");  BeanFactory factory = new XmlBeanFactory(is);    //Get bean  HelloWorld obj = (HelloWorld) factory.getBean("helloWorld"); |

Other ways to create bean factory are as below:

|  |
| --- |
| How to Create XmlBeanFactory |
| Resource resource = new FileSystemResource("beans.xml");  BeanFactory factory = new XmlBeanFactory(resource);    ClassPathResource resource = new ClassPathResource("beans.xml");  BeanFactory factory = new XmlBeanFactory(resource); |

Basically that’s all there is. Using getBean(String), you can retrieve instances of your beans; the client-side view of the BeanFactory is surprisingly simple.

The BeanFactory interface has only six methods for client code to call:

1. boolean containsBean(String): returns true if the BeanFactory contains a bean definition or bean instance that matches the given name
2. Object getBean(String): returns an instance of the bean registered under the given name. Depending on how the bean was configured by the BeanFactory configuration, either a singleton and thus shared instance or a newly created bean will be returned. A BeansException will be thrown when either the bean could not be found (in which case it’ll be a NoSuchBeanDefinitionException), or an exception occurred while instantiating and preparing the bean
3. Object getBean(String, Class): returns a bean, registered under the given name. The bean returned will be cast to the given Class. If the bean could not be cast, corresponding exceptions will be thrown (BeanNotOfRequiredTypeException). Furthermore, all rules of the getBean(String) method apply (see above)
4. Class getType(String name): returns the Class of the bean with the given name. If no bean corresponding to the given name could be found, a NoSuchBeanDefinitionException will be thrown
5. boolean isSingleton(String): determines whether or not the bean definition or bean instance registered under the given name is a singleton. If no bean corresponding to the given name could be found, a NoSuchBeanDefinitionException will be thrown
6. String[] getAliases(String): Return the aliases for the given bean name, if any were defined in the bean definition

## 2. Spring ApplicationContext container

**ApplicationContext** container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext* interface.

The *ApplicationContext* container includes all functionality of the *BeanFactory* container, so it is generally recommended over the *BeanFactory*. BeanFactory can still be used for lightweight applications like mobile devices or applet based applications where data volume and speed is significant.

The most commonly used ApplicationContext implementations are:

1. [**FileSystemXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html) – This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
2. [**ClassPathXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) – This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
3. [**WebXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/support/XmlWebApplicationContext.html) – This container loads the XML file with definitions of all beans from within a web application.

A sample code for application context instantiation will look like this.

|  |
| --- |
| How to create ApplicationContext |
| ApplicationContext context = new FileSystemXmlApplicationContext("beans.xml");  HelloWorld obj = (HelloWorld) context.getBean("helloWorld"); |

1. What is the difference between XML and Java Configurations for Spring?

If you are using a Java @Configuration file, you can simultaneously load in an XML configuration file simply by adding an @ImportResource annotation. The following annotation loads in a Spring XML [configuration file](https://searchitoperations.techtarget.com/definition/configuration-file) named meanbeans.xml into the annotation based Spring ApplicationContext object:  
 **@Configuration  
@ImportResource("classpath:meanbeans.xml")  
class JConfig { }**

So, if you've got a class named FooBean and a class named BarBean, and you have one bean configured in the XML file, and the other configured in the Java @Configuration file, you can pull them both into your application from a single beanFactory.

**class FooBean { }  
class BarBean { }**

So, this would be the goal:

**public class XmlAndAnnotations {  
  
  public static void main(String args[]) {  
    AnnotationConfigApplicationContext beanFactory =  
    new AnnotationConfigApplicationContext(JConfig.class);  
      //new ClassPathXmlApplicationContext("meanbeans.xml");  
  
    System.out.println(*beanFactory.getBean("fooBean")*);  
    System.out.println(*beanFactory.getBean("barBean")*);  
  }  
}**

Notice how both the fooBean and barBean are pulled from a common beanFactory, despite the two beans being configured in different configuration resources? Here's the full Java based configuration file, which only specifically describes the creation of the fooBean:

**@Configuration  
@ImportResource("classpath:meanbeans.xml")  
class JConfig {  
  @Bean()  
  public FooBean fooBean() {return new FooBean();}  
}**

And here's the meanbeans.xml file, which describes the barBean:

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p" xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context" xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx" xsi:schemaLocation=" http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.5.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd" default-autowire="autodetect">   <bean id="barBean" class="com.mcnz.spring.BarBean" /> </beans> |

And that's it! When the main method of the XmlAndAnnotations class runs, both the fooBean and barBean will be loaded from two separate Spring configuration resources!

1. How do you choose between XML and Java Configurations for Spring?

They are two different ways to configure a bean in spring.  
Annotation based configuration is growing popularity now a days as it is more like java.  
  
eg:  
  
**XML based:**  
*Beans.xml*  
  
<bean id="helloWorld" class="HelloWorld" >  
       <!-- collaborators and configuration for this bean go here -->  
   </bean>  
  
**in java class,**  
  
ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml"); HelloWorld objA = (HelloWorld) context.getBean("helloWorld");  
  
**Annotation Based:**  
  
**Beans.java**  
 @Bean  
   public HelloWorld helloWorld(){  
     return new HelloWorld();  
   }  
  
**in java class,**  
  
ApplicationContext ctx =     new AnnotationConfigApplicationContext(HelloWorldConfig.class);       HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

1. How does Spring do Autowiring?

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

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

## **Advantage of Autowiring**

It requires the **less code** because we don't need to write the code to inject the dependency explicitly.

## **Disadvantage of Autowiring**

No control of programmer.

It can't be used for primitive and string values.

## **Autowiring Modes**

There are many autowiring modes:

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 1) | no | It is the default autowiring mode. It means no autowiring bydefault. |
| 2) | byName | The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method. |
| 3) | byType | The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method. |
| 4) | constructor | The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters. |
| 5) | autodetect | It is deprecated since Spring 3. |

## **Example of Autowiring**

Let's see the simple code to use autowiring in spring. You need to use autowire attribute of bean element to apply the autowire modes.

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

Let's see the full example of autowiring in spring. To create this example, we have created 4 files.

1. **B.java**
2. **A.java**
3. **applicationContext.xml**
4. **Test.java**

**B.java**

This class contains a constructor and method only.

1. **package** org.sssit;
2. **public** **class** B {
3. B(){System.out.println("b is created");}
4. **void** print(){System.out.println("hello b");}
5. }

**A.java**

This class contains reference of B class and constructor and method.

1. **package** org.sssit;
2. **public** **class** A {
3. B b;
4. A(){System.out.println("a is created");}
5. **public** B getB() {
6. **return** b;
7. }
8. **public** **void** setB(B b) {
9. **this**.b = b;
10. }
11. **void** print(){System.out.println("hello a");}
12. **void** display(){
13. print();
14. b.print();
15. }
16. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="b" **class**="org.sssit.B"></bean>
10. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>
12. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the display method.

1. **package** org.sssit;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test {
5. **public** **static** **void** main(String[] args) {
6. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
7. A a=context.getBean("a",A.**class**);
8. a.display();
9. }
10. }

Output:

b is created

a is created

hello a

hello b

## **1) 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>

## **2) 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.

## **3) 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>

## **4) 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>
3. What are the different kinds of matching used by Spring for Autowiring?

Usually we provide bean configuration details in the spring bean configuration file and we also specify the beans that will be injected in other beans using ref attribute. But Spring framework provides autowiring features too where we don’t need to provide bean injection details explicitly.

There are different ways through which we can autowire a spring bean.

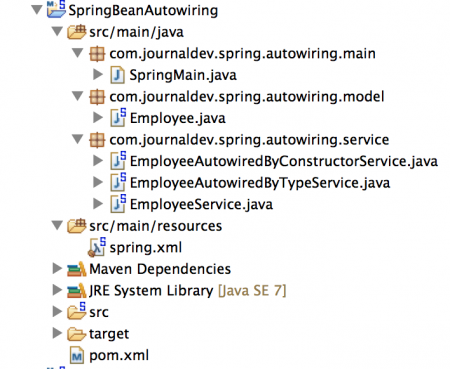
1. autowire byName – For this type of autowiring, setter method is used for dependency injection. Also the variable name should be same in the class where we will inject the dependency and in the spring bean configuration file.
2. autowire byType – For this type of autowiring, class type is used. So there should be only one bean configured for this type in the spring bean configuration file.
3. autowire by constructor – This is almost similar to autowire byType, the only difference is that constructor is used to inject the dependency.
4. autowire by autodetect – If you are on Spring 3.0 or older versions, this is one of the autowire options available. This option was used for autowire by constructor or byType, as determined by Spring container. Since we already have so many options, this option is deprecated. I will not cover this option in this tutorial.
5. @Autowired annotation – We can use Spring @Autowired annotation for spring bean autowiring. @Autowired annotation can be applied on variables and methods for autowiring byType. We can also use @Autowired annotation on constructor for constructor based spring autowiring.

For @Autowired annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by **context:annotation-config** element or by defining a bean of type org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.

1. @Qualifier annotation – This annotation is used to avoid conflicts in bean mapping and we need to provide the bean name that will be used for autowiring. This way we can avoid issues where multiple beans are defined for same type. This annotation usually works with the @Autowired annotation. For constructors with multiple arguments, we can use this annotation with the argument names in the method.

By default spring bean autowiring is turned off. Spring bean autowire default value is “default” that means no autowiring is to be performed. autowire value “no” also have the same behavior.

To showcase the use of Spring Bean autowiring, let’s create a simple Spring Maven project. Our final project will look like below image.

[](https://cdn.journaldev.com/wp-content/uploads/2014/04/Spring-bean-autowire-project.png)

Let’s look into each of the autowire options one by one. For that we will create a Model bean and a service class where we will inject the model bean.

### Spring @Autowired Annotation – Maven Dependencies

For spring autowiring, we don’t need to add any additional dependencies. Our pom.xml file has spring framework core dependencies and looks like below.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.springframework.samples</groupId>

<artifactId>SpringBeanAutowiring</artifactId>

<version>0.0.1-SNAPSHOT</version>

<properties>

<!-- Generic properties -->

<java.version>1.6</java.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<!-- Spring -->

<spring-framework.version>4.0.2.RELEASE</spring-framework.version>

<!-- Logging -->

<logback.version>1.0.13</logback.version>

<slf4j.version>1.7.5</slf4j.version>

</properties>

<dependencies>

<!-- Spring and Transactions -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring-framework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring-framework.version}</version>

</dependency>

<!-- Logging with SLF4J & LogBack -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4j.version}</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

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

<scope>runtime</scope>

</dependency>

</dependencies>

</project>

### Spring @Autowired Annotation – Model Bean

Let’s create a simple Java Bean, named Employee. This bean will have a single property with getter and setter methods. We will initialize this property value in the spring bean configuration file.

package com.journaldev.spring.autowiring.model;

public class Employee {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

### Spring @Autowired Annotation – Service Class

Let’s create our service class in which we will inject Employee bean through spring autowiring.

package com.journaldev.spring.autowiring.service;

import com.journaldev.spring.autowiring.model.Employee;

public class EmployeeService {

private Employee employee;

// constructor is used for autowire by constructor

public EmployeeService(Employee emp) {

System.out.println("Autowiring by constructor used");

this.employee = emp;

}

// default constructor to avoid BeanInstantiationException for autowire

// byName or byType

public EmployeeService() {

System.out.println("Default Constructor used");

}

// used for autowire byName and byType

public void setEmployee(Employee emp) {

this.employee = emp;

}

public Employee getEmployee() {

return this.employee;

}

}

We will use the same service class for perform spring autowiring byName, byType and by constructor. The setter method will be used for spring autowiring byName and byType whereas constructor based injection will be used by constructor autowire attribute.

When we use spring autowire byName or byType, default constructor is used. That’s why we have explicitly defined the default constructor for the EmployeeService bean.

### Spring @Autowired Annotation – autowiring byType Example

Let’s create a separate class with Spring @Autowired annotation for autowiring byType.

package com.journaldev.spring.autowiring.service;

import org.springframework.beans.factory.annotation.Autowired;

import com.journaldev.spring.autowiring.model.Employee;

public class EmployeeAutowiredByTypeService {

//Autowired annotation on variable/setters is equivalent to autowire="byType"

@Autowired

private Employee employee;

@Autowired

public void setEmployee(Employee emp){

this.employee=emp;

}

public Employee getEmployee(){

return this.employee;

}

}

Note that I have annotated both Employee variable and it’s setter method with Spring @Autowired annotation, however only one of these is sufficient for spring bean autowiring.

### Spring @Autowired Annotation and @Qualifier Bean autowiring by constructor Example

Let’s create another service class where we will use @Autowired annotation for constructor based injection. We will also see @Qualifier annotation usage.

package com.journaldev.spring.autowiring.service;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import com.journaldev.spring.autowiring.model.Employee;

public class EmployeeAutowiredByConstructorService {

private Employee employee;

//Autowired annotation on Constructor is equivalent to autowire="constructor"

@Autowired(required=false)

public EmployeeAutowiredByConstructorService(@Qualifier("employee") Employee emp){

this.employee=emp;

}

public Employee getEmployee() {

return this.employee;

}

}

When this bean will be initialized by Spring framework, bean with name as “employee” will be used for autowiring. Spring @Autowired annotation excepts one argument “required” that is a boolean with default value as TRUE. We can define it to be “false” so that spring framework don’t throw any exception if no suitable bean is found for autowiring.

### Spring @Autowired Annotation – Bean Configuration File

Spring bean configuration file is the main part of any spring application, let’s see how our spring bean configuration file looks and then we will look into each part of it.

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

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"

default-autowire="byName" default-autowire-candidates="\*" >

<bean name="employee" class="com.journaldev.spring.autowiring.model.Employee">

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

</bean>

<bean name="employee1" class="com.journaldev.spring.autowiring.model.Employee" autowire-candidate="false">

<property name="name" value="Dummy Name"></property>

</bean>

<!-- autowiring byName, bean name should be same as the property name -->

<bean name="employeeServiceByName" class="com.journaldev.spring.autowiring.service.EmployeeService" autowire="byName" />

<!-- autowiring byType, there should be only one bean definition for the mapping -->

<bean name="employeeServiceByType" class="com.journaldev.spring.autowiring.service.EmployeeService" autowire="byType" />

<!-- autowiring by constructor -->

<bean name="employeeServiceConstructor" class="com.journaldev.spring.autowiring.service.EmployeeService" autowire="constructor" />

<!-- Enable Annotation based configuration -->

<context:annotation-config />

<!-- using @Autowiring annotation in below beans, byType and constructor -->

<bean name="employeeAutowiredByTypeService" class="com.journaldev.spring.autowiring.service.EmployeeAutowiredByTypeService" />

<bean name="employeeAutowiredByConstructorService" class="com.journaldev.spring.autowiring.service.EmployeeAutowiredByConstructorService" />

</beans>

Important points about spring bean configuration file are:

* **beans** element default-autowire is used to define the default autowiring method. Here I am defining the default autowiring method to be byName.
* **beans** element default-autowire-candidates is used to provide the pattern for bean names that can be used for autowiring. For simplicity I am allowing all the bean definitions to be eligible for autowiring, however if we can define some pattern for autowiring. For example, if we want only DAO bean definitions for autowiring, we can specify it as default-autowire-candidates="\*DAO".
* autowire-candidate="false" is used in a bean definition to make it ineligible for autowiring. It’s useful when we have multiple bean definitions for a single type and we want some of them not to be autowired. For example, in above spring bean configurations “employee1” bean will not be used for autowiring.
* autowire attribute byName, byType and constructor is self understood, nothing much to explain there.
* context:annotation-config is used to enable annotation based configuration support. Notice that employeeAutowiredByTypeService and employeeAutowiredByConstructorService beans don’t have autowire attributes.

### Spring @Autowired Annotation – Test Program

Now that our spring application is ready with all types of spring autowiring, let’s write a simple test program to see if it works as expected or not.

package com.journaldev.spring.autowiring.main;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.journaldev.spring.autowiring.service.EmployeeAutowiredByConstructorService;

import com.journaldev.spring.autowiring.service.EmployeeAutowiredByTypeService;

import com.journaldev.spring.autowiring.service.EmployeeService;

public class SpringMain {

public static void main(String[] args) {

ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("spring.xml");

EmployeeService serviceByName = ctx.getBean("employeeServiceByName", EmployeeService.class);

System.out.println("Autowiring byName. Employee Name="+serviceByName.getEmployee().getName());

EmployeeService serviceByType = ctx.getBean("employeeServiceByType", EmployeeService.class);

System.out.println("Autowiring byType. Employee Name="+serviceByType.getEmployee().getName());

EmployeeService serviceByConstructor = ctx.getBean("employeeServiceConstructor", EmployeeService.class);

System.out.println("Autowiring by Constructor. Employee Name="+serviceByConstructor.getEmployee().getName());

//printing hashcode to confirm all the objects are of different type

System.out.println(serviceByName.hashCode()+"::"+serviceByType.hashCode()+"::"+serviceByConstructor.hashCode());

//Testing @Autowired annotations

EmployeeAutowiredByTypeService autowiredByTypeService = ctx.getBean("employeeAutowiredByTypeService",EmployeeAutowiredByTypeService.class);

System.out.println("@Autowired byType. Employee Name="+autowiredByTypeService.getEmployee().getName());

EmployeeAutowiredByConstructorService autowiredByConstructorService = ctx.getBean("employeeAutowiredByConstructorService",EmployeeAutowiredByConstructorService.class);

System.out.println("@Autowired by Constructor. Employee Name="+autowiredByConstructorService.getEmployee().getName());

ctx.close();

}

}

The program is simple, we are just creating the spring application context and using it to get different beans and printing the employee name.

When we run above application, we get following output.

Mar 31, 2014 10:41:58 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@3fa99295: startup date [Mon Mar 31 22:41:58 PDT 2014]; root of context hierarchy

Mar 31, 2014 10:41:58 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [spring.xml]

Default Constructor used

Default Constructor used

Autowiring by constructor used

Autowiring byName. Employee Name=Pankaj

Autowiring byType. Employee Name=Pankaj

Autowiring by Constructor. Employee Name=Pankaj

21594592::15571401::1863015320

@Autowired byType. Employee Name=Pankaj

@Autowired by Constructor. Employee Name=Pankaj

Mar 31, 2014 10:41:58 PM org.springframework.context.support.ClassPathXmlApplicationContext doClose

INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@3fa99295: startup date [Mon Mar 31 22:41:58 PDT 2014]; root of context hierarchy

As you can see that for autowire byName and byType, default no-args constructor is used to initialize the bean. For autowire by constructor, parameter based constructor is used.

From the hashcode of all the variables, we have confirmed that all the spring beans are different objects and not referring to the same object.

Since we removed “employee1” from the list of eligible beans for autowiring, there was no confusion in the bean mapping. If we remove autowire-candidate="false" from the “employee1” definition, we will get below error message when executing the above main method.

Exception in thread "main" org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'employeeServiceByType' defined in class path resource [spring.xml]: Unsatisfied dependency expressed through bean property 'employee': : No qualifying bean of type [com.journaldev.spring.autowiring.model.Employee] is defined: expected single matching bean but found 2: employee,employee1; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type [com.journaldev.spring.autowiring.model.Employee] is defined: expected single matching bean but found 2: employee,employee1

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.autowireByType(AbstractAutowireCapableBeanFactory.java:1278)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(AbstractAutowireCapableBeanFactory.java:1170)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:537)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:475)

at org.springframework.beans.factory.support.AbstractBeanFactory$1.getObject(AbstractBeanFactory.java:304)

at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:228)

at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:300)

at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:195)

at org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstantiateSingletons(DefaultListableBeanFactory.java:700)

at org.springframework.context.support.AbstractApplicationContext.finishBeanFactoryInitialization(AbstractApplicationContext.java:760)

at org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:482)

at org.springframework.context.support.ClassPathXmlApplicationContext.<init>(ClassPathXmlApplicationContext.java:139)

at org.springframework.context.support.ClassPathXmlApplicationContext.<init>(ClassPathXmlApplicationContext.java:83)

at com.journaldev.spring.autowiring.main.SpringMain.main(SpringMain.java:12)

Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type [com.journaldev.spring.autowiring.model.Employee] is defined: expected single matching bean but found 2: employee,employee1

at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultListableBeanFactory.java:967)

at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListableBeanFactory.java:855)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.autowireByType(AbstractAutowireCapableBeanFactory.java:1263)

... 13 more

1. How do you debug problems with Spring Framework?

Yes, Spring framework logging is very detailed, You did not mention in your post, if you are already using a logging framework or not. If you are using log4j then just add spring appenders to the log4j config (i.e to log4j.xml or log4j.properties), If you are using log4j xml config you can do some thing like this

<category name="org.springframework.beans">

<priority value="debug" />

</category>

or

<category name="org.springframework">

<priority value="debug" />

</category>

I would advise you to test this problem in isolation using JUnit test, You can do this by using [spring testing module](http://blog.springsource.com/2011/06/21/spring-3-1-m2-testing-with-configuration-classes-and-profiles/) in conjunction with [Junit](http://www.junit.org/). If you use spring test module it will do the bulk of the work for you it loads context file based on your context config and starts container so you can just focus on testing your business logic. I have a small example here

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations={"classpath:springContext.xml"})

@Transactional

public class SpringDAOTest

{

@Autowired

private SpringDAO dao;

@Autowired

private ApplicationContext appContext;

@Test

public void checkConfig()

{

AnySpringBean bean = appContext.getBean(AnySpringBean.class);

Assert.assertNotNull(bean);

}

}

## **UPDATE**

I am not advising you to change the way you load logging but try this in your dev environment, Add this snippet to your web.xml file

<context-param>

<param-name>log4jConfigLocation</param-name>

<param-value>/WEB-INF/log4j.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>

</listener>

**UPDATE log4j config file**

I tested this on my local tomcat and it generated a lot of logging on application start up. I also want to make a correction: use **debug** not **info** as @Rayan Stewart mentioned.

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

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/" debug="false">

<appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">

<param name="Threshold" value="debug" />

<layout class="org.apache.log4j.PatternLayout">

<param name="ConversionPattern"

value="%d{HH:mm:ss} %p [%t]:%c{3}.%M()%L - %m%n" />

</layout>

</appender>

<appender name="springAppender" class="org.apache.log4j.RollingFileAppender">

<param name="file" value="C:/tomcatLogs/webApp/spring-details.log" />

<param name="append" value="true" />

<layout class="org.apache.log4j.PatternLayout">

<param name="ConversionPattern"

value="%d{MM/dd/yyyy HH:mm:ss} [%t]:%c{5}.%M()%L %m%n" />

</layout>

</appender>

<category name="org.springframework">

<priority value="debug" />

</category>

<category name="org.springframework.beans">

<priority value="debug" />

</category>

<category name="org.springframework.security">

<priority value="debug" />

</category>

<category

name="org.springframework.beans.CachedIntrospectionResults">

<priority value="debug" />

</category>

<category name="org.springframework.jdbc.core">

<priority value="debug" />

</category>

<category name="org.springframework.transaction.support.TransactionSynchronizationManager">

<priority value="debug" />

</category>

<root>

<priority value="debug" />

<appender-ref ref="springAppender" />

<!-- <appender-ref ref="STDOUT"/> -->

</root>

</log4j:configuration>

1. How do you solve NoUniqueBeanDefinitionException?
2. When you do have more than one bean of a given type, you need to tell Spring which bean you wish it to use for dependency injection. If you fail to do so, Spring will throw a [NoUniqueBeanDefinitionException](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/NoUniqueBeanDefinitionException.html) exception, which means there’s more than one bean which would fulfill the requirement.
3. There are two simple ways you can resolve the NoUniqueBeanDefinitionException exception in Spring. You can use the @Primary  annotation, which will tell Spring when all other things are equal to select the primary bean over other instances of that type for the autowire requirement.
4. The second way, is to use the @Qualifier  annotation. Through the use of this annotation, you can give Spring hints about the name of the bean you want to use. By default, the reference name of the bean is typically the lower case class name.
5. In the video below, I go through the dependency injection example used in my Spring Core course, and show you how to modify it to get the NoUniqueBeanDefinitionException. I then walk through first using the @Primary  annotation to give a preference to one bean over the other, and the I use the @Qualifier  to specifically select which instance of the bean into my classes.
6. How do you solve NoSuchBeanDefinitionException?

84

The [javadoc of NoSuchBeanDefinitionException](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/NoSuchBeanDefinitionException.html) explains

Exception thrown when a BeanFactory is asked for a bean instance for which it cannot find a definition. This may point to a non-existing bean, a non-unique bean, or a manually registered singleton instance without an associated bean definition.

A [BeanFactory](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html) is basically the abstraction representing [Spring's Inversion of Control container](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html). It exposes beans internally and externally, to your application. When it cannot find or retrieve these beans, it throws a NoSuchBeanDefinitionException.

Below are simple reasons why a BeanFactory (or related classes) would not be able to find a bean and how you can make sure it does.

# The bean doesn't exist, it wasn't registered

In the example below

@Configuration

public class Example {

public static void main(String[] args) throws Exception {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Example.class);

ctx.getBean(Foo.class);

}

}

class Foo {}

we haven't registered a bean definition for the type Foo either through a @Bean method, @Component scanning, an XML definition, or any other way. The BeanFactory managed by the AnnotationConfigApplicationContext therefore has no indication of where to get the bean requested by getBean(Foo.class). The snippet above throws

Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException:

No qualifying bean of type [com.example.Foo] is defined

Similarly, the exception could have been thrown while trying to satisfy an @Autowired dependency. For example,

@Configuration

@ComponentScan

public class Example {

public static void main(String[] args) throws Exception {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Example.class);

}

}

@Component

class Foo { @Autowired Bar bar; }

class Bar { }

Here, a bean definition is registered for Foo through @ComponentScan. But Spring knows nothing of Bar. It therefore fails to find a corresponding bean while trying to autowire the bar field of the Foo bean instance. It throws (nested inside a [UnsatisfiedDependencyException](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/UnsatisfiedDependencyException.html))

Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:

No qualifying bean of type [com.example.Bar] found for dependency [com.example.Bar]:

expected at least 1 bean which qualifies as autowire candidate for this dependency. Dependency annotations: {@org.springframework.beans.factory.annotation.Autowired(required=true)}

There are multiple ways to register bean definitions.

* @Bean method in a @Configuration class or <bean> in XML configuration
* @Component (and its meta-annotations, eg. @Repository) through @ComponentScan or <context:component-scan ... /> in XML
* Manually through [GenericApplicationContext#registerBeanDefinition](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/GenericApplicationContext.html#registerBeanDefinition-java.lang.String-org.springframework.beans.factory.config.BeanDefinition-)
* Manually through BeanDefinitionRegistryPostProcessor

...and more.

Make sure the beans you expect are properly registered.

**A common error** is to register beans multiple times, ie. mixing the options above for the same type. For example, I might have

@Component

public class Foo {}

and an XML configuration with

<context:component-scan base-packages="com.example" />

<bean name="eg-different-name" class="com.example.Foo />

Such a configuration would register two beans of type Foo, one with name foo and another with name eg-different-name. Make sure you're not accidentally registering more beans than you wanted. Which leads us to...

If you're using both XML and annotation-based configurations, make sure you import one from the other. XML provides

<import resource=""/>

while Java provides the [@ImportResource](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ImportResource.html) annotation.

# Expected single matching bean, but found 2 (or more)

There are times when you need multiple beans for the same type (or interface). For example, your application may use two databases, a MySQL instance and an Oracle one. In such a case, you'd have two DataSource beans to manage connections to each one. For (simplified) example, the following

@Configuration

public class Example {

public static void main(String[] args) throws Exception {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Example.class);

System.out.println(ctx.getBean(DataSource.class));

}

@Bean(name = "mysql")

public DataSource mysql() { return new MySQL(); }

@Bean(name = "oracle")

public DataSource oracle() { return new Oracle(); }

}

interface DataSource{}

class MySQL implements DataSource {}

class Oracle implements DataSource {}

throws

Exception in thread "main" org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No qualifying bean of type [com.example.DataSource] is defined:

expected single matching bean but found 2: oracle,mysql

because both beans registered through @Bean methods satisfied the requirement of [BeanFactory#getBean(Class)](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html#getBean-java.lang.Class-), ie. they both implement DataSource. In this example, Spring has no mechanism to differentiate or prioritize between the two. But such mechanisms exists.

You could use [@Primary](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Primary.html) (and its equivalent in XML) as described in the [documentation](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-autowired-annotation-primary) and in [this post](https://stackoverflow.com/questions/10534053/autowiring-two-beans-implementing-same-interface-how-to-set-default-bean-to-au). With this change

@Bean(name = "mysql")

@Primary

public DataSource mysql() { return new MySQL(); }

the previous snippet would not throw the exception and would instead return the mysql bean.

You can also use @Qualifier (and its equivalent in XML) to have more control over the bean selection process, as described in the [documentation](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-autowired-annotation-qualifiers). While @Autowired is primarily used to autowire by type, @Qualifier lets you autowire by name. For example,

@Bean(name = "mysql")

@Qualifier(value = "main")

public DataSource mysql() { return new MySQL(); }

could now be injected as

@Qualifier("main") // or @Qualifier("mysql"), to use the bean name

private DataSource dataSource;

without issue. [@Resource](https://docs.oracle.com/javase/7/docs/api/javax/annotation/Resource.html) is also an option.

# Using wrong bean name

Just as there are multiple ways to register beans, there are also multiple ways to name them.

[@Bean](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html) has [name](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html#name--)

The name of this bean, or if plural, aliases for this bean. If left unspecified the name of the bean is the name of the annotated method. If specified, the method name is ignored.

<bean> has the id attribute to represent the unique identifier for a bean and name can be used to create one or more aliases illegal in an (XML) id.

[@Component](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Component.html) and its meta annotations have [value](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Component.html#value--)

The value may indicate a suggestion for a logical component name, to be turned into a Spring bean in case of an autodetected component.

If that's left unspecified, a bean name is automatically generated for the annotated type, typically the lower camel case version of the type name.

@Qualifier, as mentioned earlier, lets you add more aliases to a bean.

**Make sure you use the right name when autowiring by name.**

# More advanced cases

## **Profiles**

[Bean definition profiles](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-definition-profiles) allow you to register beans conditionally. [@Profile](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Profile.html), specifically,

Indicates that a component is eligible for registration when one or more specified profiles are active.

A profile is a named logical grouping that may be activated programmatically via [ConfigurableEnvironment.setActiveProfiles(java.lang.String...)](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/env/ConfigurableEnvironment.html#setActiveProfiles-java.lang.String...-) or declaratively by setting the spring.profiles.active property as a JVM system property, as an environment variable, or as a Servlet context parameter in web.xml for web applications. Profiles may also be activated declaratively in integration tests via the [@ActiveProfiles](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/ActiveProfiles.html) annotation.

Consider this examples where the spring.profiles.active property is not set.

@Configuration

@ComponentScan

public class Example {

public static void main(String[] args) throws Exception {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Example.class);

System.out.println(Arrays.toString(ctx.getEnvironment().getActiveProfiles()));

System.out.println(ctx.getBean(Foo.class));

}

}

@Profile(value = "StackOverflow")

@Component

class Foo {

}

This will show no active profiles and throw a NoSuchBeanDefinitionException for a Foo bean. Since the StackOverflow profile wasn't active, the bean wasn't registered.

Instead, if I initialize the ApplicationContext while registering the appropriate profile

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles("StackOverflow");

ctx.register(Example.class);

ctx.refresh();

the bean is registered and can be returned/injected.

## **AOP Proxies**

Spring uses [AOP proxies](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html#aop-introduction-proxies) a lot to implement advanced behavior. Some examples include:

* [Transaction management](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html) with [@Transactional](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html)
* [Caching](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html#cache-annotations) with [@Cacheable](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html)
* [Scheduling and asynchronous execution](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/scheduling.html#scheduling-annotation-support) with [@Async](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Async.html) and [@Scheduled](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Scheduled.html)

To achieve this, Spring has two options:

1. Use the JDK's [Proxy](https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html) class to create an instance of a dynamic class at runtime which **only implements your bean's interfaces** and delegates all method invocations to an actual bean instance.
2. Use [CGLIB](https://github.com/cglib/cglib/wiki) proxies to create an instance of a dynamic class at runtime which implements both interfaces and concrete types of your target bean and delegates all method invocations to an actual bean instance.

Take this example of JDK proxies (achieved through @EnableAsync's default proxyTargetClass of false)

@Configuration

@EnableAsync

public class Example {

public static void main(String[] args) throws Exception {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Example.class);

System.out.println(ctx.getBean(HttpClientImpl.class).getClass());

}

}

interface HttpClient {

void doGetAsync();

}

@Component

class HttpClientImpl implements HttpClient {

@Async

public void doGetAsync() {

System.out.println(Thread.currentThread());

}

}

Here, Spring attempts to find a bean of type HttpClientImpl which we expect to find because the type is clearly annotated with @Component. However, instead, we get an exception

Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException:

No qualifying bean of type [com.example.HttpClientImpl] is defined

Spring wrapped the HttpClientImpl bean and exposed it through a Proxy object that only implements HttpClient. So you could retrieve it with

ctx.getBean(HttpClient.class) // returns a dynamic class: com.example.$Proxy33

// or

@Autowired private HttpClient httpClient;

It's always recommended to [program to interfaces](https://stackoverflow.com/questions/383947/what-does-it-mean-to-program-to-an-interface). When you can't, you can tell Spring to use CGLIB proxies. For example, with [@EnableAsync](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableAsync.html), you can set [proxyTargetClass](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableAsync.html#proxyTargetClass--) to true. Similar annotations (EnableTransactionManagement, etc.) have similar attributes. XML will also have equivalent configuration options.

## **ApplicationContext Hierarchies - Spring MVC**

Spring lets you build ApplicationContext instances with other ApplicationContext instances as parents, using [ConfigurableApplicationContext#setParent(ApplicationContext)](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ConfigurableApplicationContext.html#setParent-org.springframework.context.ApplicationContext-). A child context will have access to beans in the parent context, but the opposite is not true. [This post](https://stackoverflow.com/questions/5132604/why-use-spring-applicationcontext-hierarchies) goes into detail about when this is useful, particularly in Spring MVC.

In a typical Spring MVC application, you define two contexts: one for the entire application (the root) and one specifically for the [DispatcherServlet](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) (routing, handler methods, controllers). You can get more details here:

* [Difference between applicationContext.xml and spring-servlet.xml in Spring Framework](https://stackoverflow.com/questions/3652090/difference-between-applicationcontext-xml-and-spring-servlet-xml-in-spring-frame)

It's also very well explained in the official documentation, [here](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-servlet).

**A common error** in Spring MVC configurations is to declare the WebMVC configuration in the root context with @EnableWebMvc annotated @Configuration classes or <mvc:annotation-driven /> in XML, but the [@Controller](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/Controller.html) beans in the servlet context. **Since the root context cannot reach into the servlet context to find any beans, no handlers are registered and all requests fail with 404s.** You won't see a NoSuchBeanDefinitionException, but the effect is the same.

Make sure your beans are registered in the appropriate context, ie. where they can be found by the beans registered for WebMVC (HandlerMapping, HandlerAdapter, ViewResolver, ExceptionResolver, etc.). The best solution is to properly isolate beans. The DispatcherServlet is responsible for routing and handling requests so all related beans should go into its context. The ContextLoaderListener, which loads the root context, should initialize any beans the rest of your application needs: services, repositories, etc.

# Arrays, collections, and maps

Beans of some known types are handled in special ways by Spring. For example, if you tried to inject an array of MovieCatalog into a field

@Autowired

private MovieCatalog[] movieCatalogs;

Spring will find all beans of type MovieCatalog, wrap them in an array, and inject that array. This is described in the [Spring documentation discussing @Autowired](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-autowired-annotation). Similar behavior applies to Set, List, and Collection injection targets.

For a Map injection target, Spring will also behave this way if the key type is String. For example, if you have

@Autowired

private Map<String, MovieCatalog> movies;

Spring will find all beans of type MovieCatalog and add them as values to a Map, where the corresponding key will be their bean name.

As described previously, if no beans of the requested type are available, Spring will throw a NoSuchBeanDefinitionException. Sometimes, however, you just want to declare a bean of these collection types like

@Bean

public List<Foo> fooList() {

return Arrays.asList(new Foo());

}

and inject them

@Autowired

private List<Foo> foos;

In this example, Spring would fail with a NoSuchBeanDefinitionException because there are no Foo beans in your context. But you didn't want a Foo bean, you wanted a List<Foo> bean. [Before Spring 4.3, you'd have to use @Resource](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-autowired-annotation-qualifiers)

For beans that are themselves defined as a collection/map or array type, @Resource is a fine solution, referring to the specific collection or array bean by unique name. That said, **as of 4.3**, collection/map and array types can be matched through Spring’s @Autowired type matching algorithm as well, as long as the element type information is preserved in @Bean return type signatures or collection inheritance hierarchies. In this case, qualifier values can be used to select among same-typed collections, as outlined in the previous paragraph.

This works for constructor, setter, and field injection.

@Resource

private List<Foo> foos;

// or since 4.3

public Example(@Autowired List<Foo> foos) {}

However, it will fail for @Bean methods, ie.

@Bean

public Bar other(List<Foo> foos) {

new Bar(foos);

}

Here, Spring ignores any @Resource or @Autowired annotating the method, because it's a @Bean method, and therefore can't apply the behavior described in the documentation. However, you can use Spring Expression Language (SpEL) to refer to beans by their name. In the example above, you could use

@Bean

public Bar other(@Value("#{fooList}") List<Foo> foos) {

new Bar(foos);

}

to refer to the bean named fooList and inject that.

* using the @EnableConfigurationProperties annotation helped me solve my issue – [Archmede](https://stackoverflow.com/users/6106641/archmede) [May 17 '18 at 17:50](https://stackoverflow.com/questions/39173982/what-is-a-nosuchbeandefinitionexception-and-how-do-i-fix-it#comment87810921_39173983)

1. What is @Primary?

In Spring framework, the @Primary annotation is used to give higher preference to a bean, when there are multiple beans of same type.

The @Primary annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean.

The following examples demonstrate the use of the @Primary annotation.

Consider the following User interface.

**User.java**

**package** com.boraji.tutorial.spring;

**public** **interface** **User** {

**public** **void** **doSomething**();

}

Create two beans named as AdminUser and GuestUser, which implement the User interface.

**AdminUser.java**

**package** com.boraji.tutorial.spring;

*/\*\**

*\* @author imssbora*

*\*/*

**public** **class** **AdminUser** **implements** **User** {

@Override

**public** **void** **doSomething**() {

System.out.println("Inside doSomething() method of AdminUser");

}

}

**GuestUser.java**

**package** com.boraji.tutorial.spring;

*/\*\**

*\* @author imssbora*

*\*/*

**public** **class** **GuestUser** **implements** **User** {

@Override

**public** **void** **doSomething**() {

System.out.println("Inside doSomething() method of GuestUser");

}

}

Declare the AdminUser and GuestUser beans in java based configuration class.

**AppConfig.java**

**package** com.boraji.tutorial.spring.config;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.context.annotation.Primary;

**import** com.boraji.tutorial.spring.AdminUser;

**import** com.boraji.tutorial.spring.GuestUser;

**import** com.boraji.tutorial.spring.User;

@Configuration

**public** **class** **AppConfig** {

@Bean

@Primary

**public** User **getAdminUser**() {

**return** **new** AdminUser();

}

@Bean

**public** User **getGuestUser**() {

**return** **new** GuestUser();

}

}

Now create main class and run application.

**MainApp.java**

**package** com.boraji.tutorial.spring;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**import** com.boraji.tutorial.spring.config.AppConfig;

*/\*\**

*\* @author imssbora*

*\*/*

**public** **class** **MainApp** {

**public** **static** **void** **main**(String[] args) {

AnnotationConfigApplicationContext context =

**new** AnnotationConfigApplicationContext(AppConfig.class);

User user=context.getBean(User.class);

user.doSomething();

context.close();

}

}

**Output**

Inside doSomething() method of AdminUser

1. What is @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. Following is an example to show the use of @Qualifier annotation.

## **Example**

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *Student*, *Profile* and *MainApp* under the *com.tutorialspoint* package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **Student.java** file −

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

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

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

public class Profile {

@Autowired

@Qualifier("student1")

private Student student;

public Profile(){

System.out.println("Inside Profile constructor." );

}

public void printAge() {

System.out.println("Age : " + student.getAge() );

}

public void printName() {

System.out.println("Name : " + student.getName() );

}

}

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

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

Profile profile = (Profile) context.getBean("profile");

profile.printAge();

profile.printName();

}

}

Consider the example of following configuration file **Beans.xml**

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

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xmlns:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Definition for profile bean -->

<bean id = "profile" class = "com.tutorialspoint.Profile"></bean>

<!-- Definition for student1 bean -->

<bean id = "student1" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "11"/>

</bean>

<!-- Definition for student2 bean -->

<bean id = "student2" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "2"/>

</bean>

</beans>

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

Inside Profile constructor.

Age : 11

Name : Zara

1. What is CDI (Contexts and Dependency Injection)?
2. Does Spring Support CDI?
3. Would you recommed to use CDI or Spring Annotations?
4. What are the major features in different versions of Spring?
5. What are new features in Spring Framework 4.0?
6. What are new features in Spring Framework 5.0?
7. What are important Spring Modules?
8. What are important Spring Projects?
9. What is the simplest way of ensuring that we are using single version of all Spring related dependencies?
10. Name some of the design patterns used in Spring Framework?
11. What do you think about Spring Framework?
12. Why is Spring Popular?
13. Can you give a big picture of the Spring Framework?

**Spring MVC**

1. What is Model 1 architecture?
2. What is Model 2 architecture?
3. What is Model 2 Front Controller architecture?
4. Can you show an example controller method in Spring MVC?
5. Can you explain a simple flow in Spring MVC?
6. What is a ViewResolver?
7. What is Model?
8. What is ModelAndView?
9. What is a RequestMapping?
10. What is Dispatcher Servlet?
11. How do you set up Dispatcher Servlet?
12. What is a form backing object?
13. How is validation done using Spring MVC?
14. What is BindingResult?
15. How do you map validation results to your view?
16. What are Spring Form Tags?
17. What is a Path Variable?
18. What is a Model Attribute?
19. What is a Session Attribute?
20. What is a init binder?
21. How do you set default date format with Spring?
22. Why is Spring MVC so popular?

**Spring Boot**

1. What is Spring Boot?
2. What are the important Goals of Spring Boot?
3. What are the important Features of Spring Boot?
4. Compare Spring Boot vs Spring?
5. Compare Spring Boot vs Spring MVC?
6. What is the importance of @SpringBootApplication?
7. What is Auto Configuration?
8. How can we find more information about Auto Configuration?
9. What is an embedded server? Why is it important?
10. What is the default embedded server with Spring Boot?
11. What are the other embedded servers supported by Spring Boot?
12. What are Starter Projects?
13. Can you give examples of important starter projects?
14. What is Starter Parent?
15. What are the different things that are defined in Starter Parent?
16. How does Spring Boot enforce common dependency management for all its Starter projects?
17. What is Spring Initializr?
18. What is application.properties?
19. What are some of the important things that can customized in application.properties?
20. How do you externalize configuration using Spring Boot?
21. How can you add custom application properties using Spring Boot?
22. What is @ConfigurationProperties?
23. What is a profile?
24. How do you define beans for a specific profile?
25. How do you create application configuration for a specific profile?
26. How do you have different configuration for different environments?
27. What is Spring Boot Actuator?
28. How do you monitor web services using Spring Boot Actuator?
29. How do you find more information about your application envrionment using Spring Boot?
30. What is a CommandLineRunner?

**Database Connectivity - JDBC, Spring JDBC & JPA**

1. What is Spring JDBC? How is different from JDBC?
2. What is a JdbcTemplate?
3. What is a RowMapper?
4. What is JPA?
5. What is Hibernate?
6. How do you define an entity in JPA?
7. What is an Entity Manager?
8. What is a Persistence Context?
9. How do you map relationships in JPA?
10. What are the different types of relationships in JPA?
11. How do you define One to One Mapping in JPA?
12. How do you define One to Many Mapping in JPA?
13. How do you define Many to Many Mapping in JPA?
14. How do you define a datasource in a Spring Context?
15. What is the use of persistence.xml
16. How do you configure Entity Manager Factory and Transaction Manager?
17. How do you define transaction management for Spring – Hibernate integration?

**Spring Data**

1. What is Spring Data?
2. What is the need for Spring Data?
3. What is Spring Data JPA?
4. What is a CrudRepository?
5. What is a PagingAndSortingRepository?

**Unit Testing**

1. How does Spring Framework Make Unit Testing Easy?
2. What is Mockito?
3. What is your favorite mocking framework?
4. How do you do mock data with Mockito?
5. What are the different mocking annotations that you worked with?
6. What is MockMvc?
7. What is @WebMvcTest?
8. What is @MockBean?
9. How do you write a unit test with MockMVC?
10. What is JSONAssert?
11. How do you write an integration test with Spring Boot?
12. What is @SpringBootTest?
13. What is @LocalServerPort?
14. What is TestRestTemplate?

**AOP**

1. What are cross cutting concerns?
2. How do you implement cross cutting concerns in a web application?
3. If you would want to log every request to a web application, what are the options you can think of?
4. If you would want to track performance of every request, what options can you think of?
5. What is an Aspect and Pointcut in AOP?
6. What are the different types of AOP advices?
7. What is weaving?
8. Compare Spring AOP vs AspectJ?

**SOAP Web Services**

1. What is a Web Service?
2. What is SOAP Web Service?
3. What is SOAP?
4. Waht is a SOAP Envelope?
5. What is SOAP Header and SOAP Body?
6. Can you give an example of SOAP Request and SOAP Response?
7. What is a SOAP Header? What kind of information is sent in a SOAP Header?
8. Can you give an example of a SOAP Header with Authentication information?
9. What is WSDL (Web Service Definition Language)?
10. What are the different parts of a WSDL?
11. What is Contract First Approach?
12. What is an XSD?
13. Can you give an example of an XSD?
14. What is JAXB?
15. How do you configure a JAXB Plugin?
16. What is an Endpoint?
17. Can you show an example endpoint written with Spring Web Services?
18. What is a MessageDispatcherServlet?
19. How do you configure a MessageDispatcherServlet?
20. How do you generate a WSDL using Spring Web Services?
21. How do you implement error handling for SOAP Web Services?
22. What is a SOAP Fault?

**RESTful Web Services**

1. What is REST?
2. What are the key concepts in designing RESTful API?
3. What are the Best Practices of RESTful Services?
4. Can you show the code for an example Get Resource method with Spring REST?
5. What happens when we return a bean from a Request Mapping Method?
6. What is GetMapping and what are the related methods available in Spring MVC?
7. Can you show the code for an example Post Resource method with Spring REST?
8. What is the appropriate HTTP Response Status for successful execution of a Resource Creation?
9. Why do we use ResponseEntity in a RESTful Service?
10. What is HATEOAS?
11. Can you give an Example Response for HATEOAS?
12. How do we implement it using Spring?
13. How do you document RESTful web services?
14. Can you give a brief idea about Swagger Documentation?
15. How do you automate generation of Swagger Documentation from RESTful Web Services?
16. How do you add custom information to Swagger Documentation generated from RESTful Web Services?
17. What is Swagger-UI?
18. What is "Representation" of a Resource?
19. What is Content Negotiation?
20. Which HTTP Header is used for Content Negotiation?
21. How do we implement it using Spring Boot?
22. How do you add XML support to your RESTful Services built with Spring Boot?
23. How do you implement Exception Handling for RESTFul Web Services?
24. What are the best practices related to Exception Handling with respect to RESTful Web Services?
25. What are the different error status that you would return in RESTful Web Services?
26. How would you implement them using Spring Boot?
27. What HTTP Response Status do you return for validation errors?
28. How do you handle Validation Errors with RESTful Web Services?
29. Why do we need Versioning for RESTful Web Services?
30. What are the versioning options that are available?
31. How do you implement Versioning for RESTful Web Services?