**Sp~~ri~~ng**  
 **®**

**Framework**

**Notes for Professionals**



**50+ pages**

of professional hints and tricks

|  |  |
| --- | --- |
| **[GoalKicker.com](https://goalkicker.com)** [Free Programming Books](https://goalkicker.com) | **Disclaimer** This is an unocial free book created for educational purposes and is not aliated with ocial Spring® Framework group(s) or company(s).  All trademarks and registered trademarks are the property of their respective owners |

**Contents**

**About** ................................................................................................................................................................................... 1

**Chapter 1: Getting started with Spring Framework** .................................................................................... 2

Section 1.1: Setup (XML Configuration) ........................................................................................................................ 2

Section 1.2: Showcasing Core Spring Features by example ..................................................................................... 3

Section 1.3: What is Spring Framework, why should we go for it? ........................................................................... 6

**Chapter 2: Spring Core** ............................................................................................................................................... 8

Section 2.1: Introduction to Spring Core ...................................................................................................................... 8

Section 2.2: Understanding How Spring Manage Dependency? ............................................................................. 9

**Chapter 3: Spring Expression Language (SpEL)** ......................................................................................... 12

Section 3.1: Syntax Reference .................................................................................................................................... 12

**Chapter 4: Obtaining a SqlRowSet from SimpleJdbcCall** ..................................................................... 13

Section 4.1: SimpleJdbcCall creation ......................................................................................................................... 13

Section 4.2: Oracle Databases ................................................................................................................................... 14

**Chapter 5: Creating and using beans** .............................................................................................................. 16

Section 5.1: Autowiring all beans of a specific type ................................................................................................. 16

Section 5.2: Basic annotation autowiring ................................................................................................................. 17

Section 5.3: Using FactoryBean for dynamic bean instantiation ........................................................................... 18

Section 5.4: Declaring Bean ....................................................................................................................................... 19

Section 5.5: Autowiring specific bean instances with @Qualifier ........................................................................... 20

Section 5.6: Autowiring specific instances of classes using generic type parameters ........................................ 21

Section 5.7: Inject prototype-scoped beans into singletons ................................................................................... 22

**Chapter 6: Bean scopes** ........................................................................................................................................... 25

Section 6.1: Additional scopes in web-aware contexts ............................................................................................ 25

Section 6.2: Prototype scope ..................................................................................................................................... 26

Section 6.3: Singleton scope ....................................................................................................................................... 28

**Chapter 7: Conditional bean registration in Spring** .................................................................................. 30

Section 7.1: Register beans only when a property or value is specified ................................................................ 30

Section 7.2: Condition annotations ............................................................................................................................ 30

**Chapter 8: Spring JSR 303 Bean Validation** .................................................................................................. 32

Section 8.1: @Valid usage to validate nested POJOs .............................................................................................. 32

Section 8.2: Spring JSR 303 Validation - Customize error messages ................................................................... 32

Section 8.3: JSR303 Annotation based validations in Springs examples .............................................................. 34

**Chapter 9: ApplicationContext Configuration** .............................................................................................. 37

Section 9.1: Autowiring ................................................................................................................................................ 37

Section 9.2: Bootstrapping the ApplicationContext ................................................................................................. 37

Section 9.3: Java Configuration ................................................................................................................................. 38

Section 9.4: Xml Configuration ................................................................................................................................... 40

**Chapter 10: RestTemplate** ..................................................................................................................................... 43

Section 10.1: Downloading a Large File ..................................................................................................................... 43

Section 10.2: Setting headers on Spring RestTemplate request ............................................................................ 43

Section 10.3: Generics results from Spring RestTemplate ...................................................................................... 44

Section 10.4: Using Preemptive Basic Authentication with RestTemplate and HttpClient .................................. 44

Section 10.5: Using Basic Authentication with HttpComponent's HttpClient ......................................................... 46

**Chapter 11: Task Execution and Scheduling** .................................................................................................. 47

Section 11.1: Enable Scheduling ................................................................................................................................... 47

Section 11.2: Cron expression ...................................................................................................................................... 47

Section 11.3: Fixed delay .............................................................................................................................................. 49

Section 11.4: Fixed Rate ............................................................................................................................................... 49

**Chapter 12: Spring Lazy Initialization** ............................................................................................................... 50

Section 12.1: Example of Lazy Init in Spring .............................................................................................................. 50

Section 12.2: For component scanning and auto-wiring ......................................................................................... 51

Section 12.3: Lazy initialization in the configuration class ....................................................................................... 51

**Chapter 13: Property Source** ................................................................................................................................. 52

Section 13.1: Sample xml configuration using PropertyPlaceholderConfigurer .................................................... 52

Section 13.2: Annotation .............................................................................................................................................. 52

**Chapter 14: Dependency Injection (DI) and Inversion of Control (IoC)** ........................................... 53

Section 14.1: Autowiring a dependency through Java configuration ..................................................................... 53

Section 14.2: Autowiring a dependency through XML configuration ..................................................................... 53

Section 14.3: Injecting a dependency manually through XML configuration ........................................................ 54

Section 14.4: Injecting a dependency manually through Java configuration ...................................................... 56

**Chapter 15: JdbcTemplate** ..................................................................................................................................... 57

Section 15.1: Basic Query methods ............................................................................................................................ 57

Section 15.2: Query for List of Maps .......................................................................................................................... 57

Section 15.3: SQLRowSet ............................................................................................................................................. 58

Section 15.4: Batch operations ................................................................................................................................... 58

Section 15.5: NamedParameterJdbcTemplate extension of JdbcTemplate ........................................................ 59

**Chapter 16: SOAP WS Consumption** ................................................................................................................... 60

Section 16.1: Consuming a SOAP WS with Basic auth .............................................................................................. 60

**Chapter 17: Spring profile** ....................................................................................................................................... 61

Section 17.1: Spring Profiles allows to configure parts available for certain environment .................................. 61

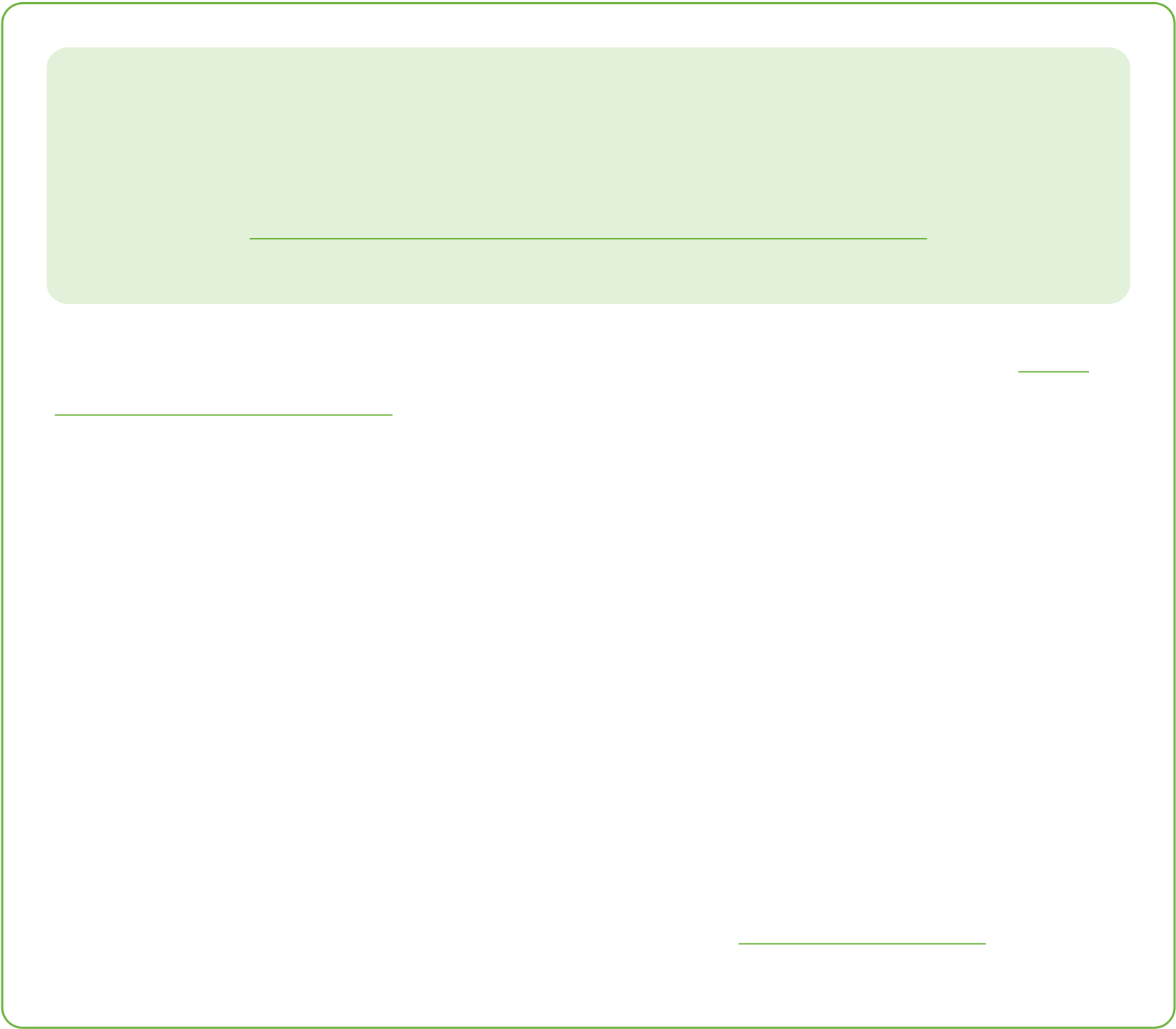
**Chapter 18: Understanding the dispatcher-servlet.xml** .......................................................................... 62

Section 18.1: dispatcher-servlet.xml ........................................................................................................................... 62

Section 18.2: dispatcher servlet configuration in web.xml ....................................................................................... 62

**Credits** .............................................................................................................................................................................. 64

**You may also like** ........................................................................................................................................................ 65



**About**

Please feel free to share this PDF with anyone for free, latest version of this book can be downloaded from: <https://goalkicker.com/SpringFrameworkBook>

This *Spring® Framework Notes for Professionals* book is compiled from [Stack Overflow Documentatio](https://archive.org/details/documentation-dump.7z)n, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Spring® Framework group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to [web@petercv.com](mailto:web@petercv.com)

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 1 |

**Chapter 1: Getting started with Spring**

**Framework**

**Version Release Date**

[5.0.x](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#new-in-5.0.1)  2017-10-24

[4.3.x](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#new-in-4.3)  2016-06-10

[4.2.x](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#new-in-4.2)  2015-07-31

[4.1.x](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#new-in-4.1)  2014-09-14

[4.0.x](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#new-in-4.0)  2013-12-12

[3.2.x](http://docs.spring.io/spring-framework/docs/3.2.x/changelog.txt)  2012-12-13

[3.1.x](http://docs.spring.io/spring-framework/docs/3.1.x/changelog.txt)  2011-12-13

[3.0.x](http://docs.spring.io/spring-framework/docs/3.0.x/changelog.txt)  2009-12-17

[2.5.x](http://docs.spring.io/spring-framework/docs/2.5.x/changelog.txt)  2007-12-25

[2.0.x](http://docs.spring.io/spring-framework/docs/2.0.x/changelog.txt)  2006-10-04

[1.2.x](http://docs.spring.io/spring-framework/docs/1.2.x/changelog.txt)  2005-05-13

[1.1.x](http://docs.spring.io/spring-framework/docs/1.1.x/changelog.txt)  2004-09-05

1.0.x 2003-03-24

**Section 1.1: Setup (XML Configuration)**

Steps to create Hello Spring:

1. Investigate Spring Boot to see if that would better suit your needs.

2. Have a project set up with the correct dependencies. It is recommended that you are using Maven or Gradle.

3. create a POJO class, e.g. Employee.java

4. create a XML file where you can define your class and variables. e.g beans.xml

5. create your main class e.g. Customer.java

6. Include [spring-beans](http://search.maven.org/#artifactdetails%7Corg.springframework%7Cspring-beans%7C4.3.1.RELEASE%7Cjar) (and its transitive dependencies!) as a dependency.

Employee.java:

|  |
| --- |
| **package**com.test; |

|  |
| --- |
| **publicclass** Employee { |

|  |
| --- |
| **private**String name; |

|  |
| --- |
| **public**String getName(){ |

**return** name;

|  |
| --- |
| } |

|  |
| --- |
| **publicvoid** setName(String name){ |

**this**.name= name;

|  |
| --- |
| } |

|  |
| --- |
| **publicvoid** displayName(){ |

System.out.println(name);

}

}

beans.xml:

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 2 |

|  |
| --- |
| **<?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-4.3.xsd"**>** |

|  |
| --- |
| **<bean**id="employee"class="com.test.Employee"**>** |

|  |
| --- |
| **<property**name="name"value="test spring"**></property>** |

|  |
| --- |
| **</bean>** |

|  |
| --- |
| **</beans>** |

Customer.java:

|  |
| --- |
| **package**com.test; |

|  |
| --- |
| **import**org.springframework.context.ApplicationContext; |

|  |
| --- |
| **import**org.springframework.context.support.ClassPathXmlApplicationContext; |

|  |
| --- |
| **publicclass** Customer { |

**publicstaticvoid** main(String[] args){

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

|  |
| --- |
| Employee obj =(Employee) context.getBean("employee"); |

obj.displayName();   
}

|  |
| --- |
| } |

**Section 1.2: Showcasing Core Spring Features by example**

**Description**

This is a self-contained running example including/showcasing: minimum *dependencies* needed, Java *Configuration*, *Bean declaration* by annotation and Java Configuration, *Dependency Injection* by Constructor and by Property, and *Pre/Post* hooks.

**Dependencies**

These dependencies are needed in the classpath:

1. [spring-core](http://search.maven.org/#artifactdetails%7Corg.springframework%7Cspring-core%7C4.3.1.RELEASE%7Cjar)   
2. [spring-context](http://search.maven.org/#artifactdetails%7Corg.springframework%7Cspring-context%7C4.3.1.RELEASE%7Cjar)   
3. [spring-beans](http://search.maven.org/#artifactdetails%7Corg.springframework%7Cspring-beans%7C4.3.1.RELEASE%7Cjar)   
4. [spring-aop](http://search.maven.org/#artifactdetails%7Corg.springframework%7Cspring-aop%7C4.3.1.RELEASE%7Cjar)   
5. [spring-expression](http://search.maven.org/#artifactdetails%7Corg.springframework%7Cspring-expression%7C4.3.1.RELEASE%7Cjar)   
6. [commons-logging](http://search.maven.org/#artifactdetails%7Ccommons-logging%7Ccommons-logging%7C1.2%7Cjar)

**Main Class**

Starting from the end, this is our Main class that serves as a placeholder for the main() method which initialises the Application Context by pointing to the Configuration class and loads all the various beans needed to showcase particular functionality.

|  |
| --- |
| **package**com.stackoverflow.documentation; |

|  |  |
| --- | --- |
| **import**org.springframework.context.ApplicationContext; |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 3 |

|  |
| --- |
| **import**org.springframework.context.annotation.AnnotationConfigApplicationContext; |

|  |
| --- |
| **publicclass** Main { |

**publicstaticvoid** main(String[] args){

*//initializing the Application Context once per application.*

ApplicationContext applicationContext =   
 **new** AnnotationConfigApplicationContext(AppConfig.**class**);

*//bean registered by annotation*  
 BeanDeclaredByAnnotation beanDeclaredByAnnotation =  
 applicationContext.getBean(BeanDeclaredByAnnotation.**class**); beanDeclaredByAnnotation.sayHello();

*//bean registered by Java configuration file*  
 BeanDeclaredInAppConfig beanDeclaredInAppConfig =  
 applicationContext.getBean(BeanDeclaredInAppConfig.**class**); beanDeclaredInAppConfig.sayHello();

*//showcasing constructor injection*  
 BeanConstructorInjection beanConstructorInjection =  
 applicationContext.getBean(BeanConstructorInjection.**class**); beanConstructorInjection.sayHello();

*//showcasing property injection*  
 BeanPropertyInjection beanPropertyInjection =  
 applicationContext.getBean(BeanPropertyInjection.**class**); beanPropertyInjection.sayHello();

*//showcasing PreConstruct / PostDestroy hooks*  
 BeanPostConstructPreDestroy beanPostConstructPreDestroy =  
 applicationContext.getBean(BeanPostConstructPreDestroy.**class**); beanPostConstructPreDestroy.sayHello();   
 }   
}

**Application Configuration file**

The configuration class is annotated by @Configuration and is used as a parameter in the initialised Application Context. The @ComponentScan annotation at the class level of the configuration class points to a package to be scanned for Beans and dependencies registered using annotations. Finally the @Bean annotation serves as a bean definition in the configuration class.

|  |
| --- |
| **package**com.stackoverflow.documentation; |

|  |
| --- |
| **import**org.springframework.context.annotation.Bean; |

**import**org.springframework.context.annotation.ComponentScan;

|  |
| --- |
| **import**org.springframework.context.annotation.Configuration; |

|  |
| --- |
| @Configuration |

@ComponentScan("com.stackoverflow.documentation")

|  |
| --- |
| **publicclass** AppConfig { |

|  |
| --- |
| @Bean |

**public** BeanDeclaredInAppConfig beanDeclaredInAppConfig(){ **returnnew** BeanDeclaredInAppConfig();

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 4 |

|  |
| --- |
| } |

**Bean Declaration by Annotation**

The @Component annotation serves to demarcate the POJO as a Spring bean available for registration during

component scanning.

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** BeanDeclaredByAnnotation { |

|  |
| --- |
| **publicvoid** sayHello(){ |

|  |
| --- |
| System.out.println("Hello, World from BeanDeclaredByAnnotation !"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Bean Declaration by Application Configuration**

Notice that we don't need to annotate or otherwise mark our POJO since the bean declaration/definition is

happening in the Application Configuration class file.

|  |
| --- |
| **publicclass** BeanDeclaredInAppConfig { |

|  |
| --- |
| **publicvoid** sayHello(){ |

|  |
| --- |
| System.out.println("Hello, World from BeanDeclaredInAppConfig !"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Constructor Injection**

Notice that the @Autowired annotation is set at the constructor level. Also notice that unless explicitely defined by

name the default autowiring is happening *based on the type* of the bean (in this instance BeanToBeInjected).

|  |
| --- |
| **package**com.stackoverflow.documentation; |

|  |
| --- |
| **import**org.springframework.beans.factory.annotation.Autowired; |

|  |
| --- |
| **import**org.springframework.stereotype.Component; |

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** BeanConstructorInjection { |

|  |
| --- |
| **private** BeanToBeInjected dependency; |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **public** BeanConstructorInjection(BeanToBeInjected dependency){ |

|  |
| --- |
| **this**.dependency= dependency; |

|  |
| --- |
| } |

|  |
| --- |
| **publicvoid** sayHello(){ |

|  |
| --- |
| System.out.print("Hello, World from BeanConstructorInjection with dependency: "); |

dependency.sayHello();

}

|  |
| --- |
| } |

**Property Injection**

Notice that the @Autowired annotation demarcates the setter method whose name follows the JavaBeans standard.

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 5 |

|  |
| --- |
| **package**com.stackoverflow.documentation; |

|  |
| --- |
| **import**org.springframework.beans.factory.annotation.Autowired; |

|  |
| --- |
| **import**org.springframework.stereotype.Component; |

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** BeanPropertyInjection { |

|  |
| --- |
| **private** BeanToBeInjected dependency; |

|  |
| --- |
| @Autowired |

**publicvoid** setBeanToBeInjected(BeanToBeInjected beanToBeInjected){ **this**.dependency= beanToBeInjected;

|  |
| --- |
| } |

|  |
| --- |
| **publicvoid** sayHello(){ |

System.out.println("Hello, World from BeanPropertyInjection !"); }   
}

**PostConstruct / PreDestroy hooks**

We can intercept initialisation and destruction of a Bean by the @PostConstruct and @PreDestroy hooks.

|  |
| --- |
| **package**com.stackoverflow.documentation; |

|  |
| --- |
| **import**org.springframework.stereotype.Component; |

**import**javax.annotation.PostConstruct;   
**import**javax.annotation.PreDestroy;

@Component   
**publicclass** BeanPostConstructPreDestroy {

@PostConstruct   
 **publicvoid** pre(){   
 System.out.println("BeanPostConstructPreDestroy - PostConstruct"); }

**publicvoid** sayHello(){   
 System.out.println(" Hello World, BeanPostConstructPreDestroy !"); }

@PreDestroy   
 **publicvoid** post(){   
 System.out.println("BeanPostConstructPreDestroy - PreDestroy"); }   
}

**Section 1.3: What is Spring Framework, why should we go for it?**

Spring is a framework, which provides bunch of classes, by using this we don't need to write boiler plate logic in our code, so Spring provides an abstract layer on J2ee.

For Example in Simple JDBC Application programmer is responsible for

|  |  |  |
| --- | --- | --- |
| 1. | Loading the driver class |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | | 6 |

2. Creating the connection   
3. Creating statement object   
4. Handling the exceptions   
5. Creating query   
6. Executing query   
7. Closing the connection

Which is treated as boilerplate code as every programmer write the same code. So for simplicity the framework takes care of boilerplate logic and the programmer has to write only business logic. So by using Spring framework we can develop projects rapidly with minimum lines of code, without any bug, the development cost and time also reduced.

**So Why to choose Spring as struts is there**

Strut is a framework which provide solution to web aspects only and struts is invasive in nature. Spring has many features over struts so we have to choose Spring.

1. Spring is ***Noninvasive*** in nature: That means you don't need to extend any classes or implement any interfaces to your class.

2. Spring is ***versatile***: That means it can integrated with any existing technology in your project.

3. Spring provides ***end to end*** project development: That means we can develop all the modules like business layer, persistence layer.

4. Spring is ***light weight***: That means if you want to work on particular module then , you don't need to learn complete spring, only learn that particular module(eg. Spring Jdbc, Spring DAO)   
5. Spring supports ***dependency injection***.

6. Spring supports ***multiple project development*** eg: Core java Application, Web Application, Distributed Application, Enterprise Application.

7. Spring supports Aspect oriented Programming for cross cutting concerns.

So finally we can say Spring is an alternative to Struts. But Spring is not a replacement of J2EE API, As Spring supplied classes internally uses J2EE API classes. Spring is a vast framework so it has divided into several modules.

No module is dependent to another except Spring Core. Some Important modules are

|  |  |  |
| --- | --- | --- |
| 1. | Spring Core |  |
| 2. | Spring JDBC |
| 3. | Spring AOP |
| 4. | Spring Transaction |
| 5. | Spring ORM |
| 6. | Spring MVC |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | | 7 |

**Chapter 2: Spring Core**

**Section 2.1: Introduction to Spring Core**

Spring is a vast framework, so the Spring framework has been divided in several modules which makes spring lightweight. Some important modules are:

1. Spring Core   
2. Spring AOP   
3. Spring JDBC   
4. Spring Transaction   
5. Spring ORM   
6. Spring MVC

All the modules of Spring are independent of each other except Spring Core. As Spring core is the base module, so in all module we have to use Spring Core

***Spring Core***

Spring Core talking all about dependency management.That means if any arbitrary classes provided to spring then Spring can manage dependency.

**What is a dependency:**   
From project point of view, in a project or application multiple classes are there with different functionality. and each classes required some functionality of other classes.

**Example:**

|  |
| --- |
| **class** Engine { |

|  |
| --- |
| **publicvoid** start(){ |

|  |
| --- |
| System.out.println("Engine started"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

|  |
| --- |
| **class** Car { |

|  |
| --- |
| **publicvoid** move(){ |

|  |
| --- |
| *// For moving start() method of engine class is required* |

}

|  |
| --- |
| } |

Here class Engine is required by class car so we can say class engine is dependent to class Car, So instead of we managing those dependency by Inheritance or creating object as fallows.

***By Inheritance:***

|  |
| --- |
| **class** Engine { |

|  |
| --- |
| **publicvoid** start(){ |

|  |
| --- |
| System.out.println("Engine started"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| **class** Car **extends** Engine { |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 8 |

|  |
| --- |
| **publicvoid** move(){ |

|  |
| --- |
| start();*//Calling super class start method,* |

|  |
| --- |
| } |

|  |
| --- |
| } |

***By creating object of dependent class:***

|  |
| --- |
| **class** Engine { |

|  |
| --- |
| **publicvoid** start(){ |

System.out.println("Engine started");

}

|  |
| --- |
| } |

|  |
| --- |
| **class** Car { |

|  |
| --- |
| Engine eng =**new** Engine(); |

|  |
| --- |
| **publicvoid** move(){ |

eng.start();

}

}

So instead of we managing dependency between classes spring core takes the responsibility dependency

management. But Some rule are there, The classes must be designed with some design technique that is Strategy

design pattern.

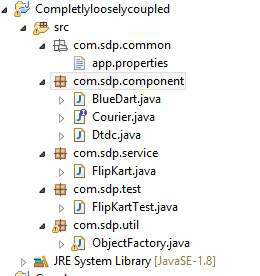
**Section 2.2: Understanding How Spring Manage Dependency?**

Let me write a piece of code which shows completely loosely coupled, Then you can easily understand how Spring

core manage the dependency internally. Consider an scenario, Online business Flipkart is there, it uses some times

DTDC or Blue Dart courier service , So let me design a application which shows complete loosely coupled. The

Eclipse Directory as fallows:



|  |
| --- |
| *//Interface* |

|  |
| --- |
| **package**com.sdp.component; |

|  |
| --- |
| **publicinterface** Courier { |

|  |
| --- |
| **public**String deliver(String iteams,String address); |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 9 |

//implementation classes

|  |
| --- |
| **package**com.sdp.component; |

|  |
| --- |
| **publicclass** BlueDart **implements** Courier { |

|  |
| --- |
| **public**String deliver(String iteams, String address){ |

|  |
| --- |
| **return** iteams+"Shiped to Address "+address +"Through BlueDart"; |

|  |
| --- |
| } |

|  |
| --- |
| } |

|  |
| --- |
| **package**com.sdp.component; |

|  |
| --- |
| **publicclass** Dtdc **implements** Courier { |

|  |
| --- |
| **public**String deliver(String iteams, String address){ |

|  |
| --- |
| **return** iteams+"Shiped to Address "+address +"Through Dtdc";} |

|  |
| --- |
| } |

//Component classe

|  |
| --- |
| **package**com.sdp.service; |

|  |
| --- |
| **import**com.sdp.component.Courier; |

|  |
| --- |
| **publicclass** FlipKart { |

|  |
| --- |
| **private** Courier courier; |

|  |
| --- |
| **publicvoid** setCourier(Courier courier){ |

**this**.courier= courier;   
}   
**publicvoid** shopping(String iteams,String address) {   
 String status=courier.deliver(iteams, address); System.out.println(status);

|  |
| --- |
| } |

|  |
| --- |
| } |

//Factory classes to create and return Object

|  |
| --- |
| **package**com.sdp.util; |

|  |
| --- |
| **import**java.io.IOException; |

|  |
| --- |
| **import**java.util.Properties; |

|  |
| --- |
| **import**com.sdp.component.Courier; |

|  |
| --- |
| **publicclass**ObjectFactory{ |

|  |
| --- |
| **privatestatic**Properties props; |

|  |
| --- |
| **static**{ |

|  |
| --- |
| props=**new**Properties(); |

|  |  |
| --- | --- |
| **try**{ |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 10 |

props.load(ObjectFactory.**class**.getClassLoader().getResourceAsStream("com//sdp//common//app.properti es"));   
 }**catch**(IOException e){   
 *// TODO Auto-generated catch block*  
 e.printStackTrace();   
 }

}   
**publicstatic**Object getInstance(String logicalclassName)   
{   
 Object obj =**null**;   
 String originalclassName=props.getProperty(logicalclassName); **try**{  
 obj=**Class**.forName(originalclassName).newInstance(); }**catch**(InstantiationException e){   
 *// TODO Auto-generated catch block*  
 e.printStackTrace();   
 }**catch**(IllegalAccessException e){   
 *// TODO Auto-generated catch block*  
 e.printStackTrace();   
 }**catch**(ClassNotFoundException e){   
 *// TODO Auto-generated catch block*  
 e.printStackTrace();   
 }   
 **return** obj;   
}

}

//properties file

|  |
| --- |
| BlueDart.**class**=com.sdp.component.BlueDart |

|  |
| --- |
| Dtdc.**class**=com.sdp.component.Dtdc |

|  |
| --- |
| FlipKart.**class**=com.sdp.service.FlipKart |

//Test class

|  |
| --- |
| **package**com.sdp.test; |

|  |
| --- |
| **import**com.sdp.component.Courier; |

**import**com.sdp.service.FlipKart;

|  |
| --- |
| **import**com.sdp.util.ObjectFactory; |

|  |
| --- |
| **publicclass** FlipKartTest { |

**publicstaticvoid** main(String[] args){  
 Courier courier=(Courier)ObjectFactory.getInstance("Dtdc.class"); FlipKart flipkart=(FlipKart)ObjectFactory.getInstance("FlipKart.class"); flipkart.setCourier(courier);

|  |
| --- |
| flipkart.shopping("Hp Laptop", "SR Nagar,Hyderabad"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

If we write this code then we can manually achieve loose coupling,this is applicable if all the classes want either BlueDart or Dtdc , But if some class want BlueDart and some other class want Dtdc then again it will be tightly coupled, So instead of we creating and managing the dependency injection Spring core takes the responsibility of creating and managing the beans, Hope This will helpful, in next example we wil see the !st application on Spring core with deitals

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 11 |

**Chapter 3: Spring Expression Language (SpEL)**

**Section 3.1: Syntax Reference**

You can use @Value("#{expression}") to inject value at runtime, in which the expression is a SpEL expression.

**Literal expressions**

Supported types include strings, dates, numeric values (int, real, and hex), boolean and null.

|  |
| --- |
| "#{'Hello World'}"*//strings* |

|  |
| --- |
| "#{3.1415926}"*//numeric values (double)* |

|  |
| --- |
| "#{true}"*//boolean* |

|  |
| --- |
| "#{null}"*//null* |

**Inline list**

|  |
| --- |
| "#{1,2,3,4}"*//list of number* |

|  |
| --- |
| "#{{'a','b'},{'x','y'}}"*//list of list* |

**Inline Maps**

|  |
| --- |
| "#{name:'Nikola',dob:'10-July-1856'}" |

|  |
| --- |
| "#{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}"*//map of maps* |

**Invoking Methods**

|  |
| --- |
| "#{'abc'.length()}"*//evaluates to 3* |

|  |
| --- |
| "#{f('hello')}"*//f is a method in the class to which this expression belongs, it has a string* |

|  |  |
| --- | --- |
| *parameter* |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 12 |

**Chapter 4: Obtaining a SqlRowSet from SimpleJdbcCall**

This describes how to directly obtain a **SqlRowSet** using **SimpleJdbcCall** with a stored procedure in your database that has a **cursor output parameter**,

I am working with an Oracle database, I've attempted to create an example that should work for other databases, my Oracle example details issues with Oracle.

**Section 4.1: SimpleJdbcCall creation**

Typically, you will want to create your SimpleJdbcCalls in a Service.

This example assumes your procedure has a single output parameter that is a cursor; you will need to adjust your declareParameters to match your procedure.

@Service   
**publicclass** MyService(){

@Autowired   
 **private** DataSource dataSource;

*// Autowire your configuration, for example*  
 @Value("${db.procedure.schema}")   
 String schema;

|  |
| --- |
| **private** SimpleJdbcCall myProcCall; |

*// create SimpleJdbcCall after properties are configured* @PostConstruct   
 **void** initialize(){   
 **this**.myProcCall=**new** SimpleJdbcCall(dataSource)  
 .withProcedureName("my\_procedure\_name") .withCatalogName("my\_package")  
 .withSchemaName(schema)  
 .declareParameters(**new** SqlOutParameter( "out\_param\_name",   
 Types.REF\_CURSOR,   
 **new** SqlRowSetResultSetExtractor())); }

**public** SqlRowSet myProc(){  
 Map<String, Object> out =**this**.myProcCall.execute(); **return**(SqlRowSet) out.get("out\_param\_name");   
}

}

There are many options you can use here:

 **withoutProcedureColumnMetaDataAccess()** needed if you have overloaded procedure names or just don't want SimpleJdbcCall to validate against the database.

 **withReturnValue()** if procedure has a return value. First value given to declareParameters defines the return value. Also, if your procedure is a function, use **withFunctionName** and **executeFunction** when executing.

|  |  |  |
| --- | --- | --- |
|  | **withNamedBinding()** if you want to give arguments using names instead of position. |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | | 13 |

 **useInParameterNames()** defines the argument order. I think this may be required if you pass in your arguments as a list instead of a map of argument name to value. Though it may only be required if you use withoutProcedureColumnMetaDataAccess()

**Section 4.2: Oracle Databases**

Here's how to resolve issues with Oracle.

Assuming your procedure output parameter is ref cursor, you will get this exception.

java.sql.SQLException: Invalid column type: 2012

So change Types.REF\_CURSOR to OracleTypes.CURSOR in **simpleJdbcCall.declareParameters()**

**Supporting OracleTypes**

*You may only need to do this if you have certain column types in your data.*

The next issue I encountered was that proprietary Types such as oracle.sql.TIMESTAMPTZ caused this error in SqlRowSetResultSetExtractor:

Invalid SQL type for column; nested exception is java.sql.SQLException: Invalid SQL type for column

So we need to create a **ResultSetExtractor** that supports Oracle types. *I will explain the reason for password after this code.*

**package**com.boost.oracle;

**import**oracle.jdbc.rowset.OracleCachedRowSet;   
**import**org.springframework.dao.DataAccessException;   
**import**org.springframework.jdbc.core.ResultSetExtractor;   
**import**org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet; **import**org.springframework.jdbc.support.rowset.SqlRowSet;

**import**java.sql.ResultSet;   
**import**java.sql.SQLException;

***/\*\****  
 ***\* OracleTypes can cause {@link org.springframework.jdbc.core.SqlRowSetResultSetExtractor} \* to fail due to a Oracle SQL type that is not in the standard {@link java.sql.Types}.***

***\****  
 ***\* Also, types such as {@link oracle.sql.TIMESTAMPTZ} require a Connection when processing \* the ResultSet; {@link OracleCachedRowSet#getConnectionInternal()} requires a JNDI \* DataSource name or the username and password to be set.***

***\****  
 ***\* For now I decided to just set the password since changing SpringBoot to a JNDI DataSource \* configuration is a bit complicated.***

***\****  
 ***\* Created by Arlo White on 2/23/17.***

***\*/***   
**publicclass** OracleSqlRowSetResultSetExtractor **implements** ResultSetExtractor<SqlRowSet>{

|  |
| --- |
| **private**String oraclePassword; |

|  |  |
| --- | --- |
| **public** OracleSqlRowSetResultSetExtractor(String oraclePassword){ |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 14 |

**this**.oraclePassword= oraclePassword;

|  |
| --- |
| } |

|  |
| --- |
| @Override |

**public** SqlRowSet extractData(ResultSet rs)**throws**SQLException, DataAccessException { OracleCachedRowSet cachedRowSet =**new** OracleCachedRowSet();   
 *// allows getConnectionInternal to get a Connection for TIMESTAMPTZ*  
 cachedRowSet.setPassword(oraclePassword);  
 cachedRowSet.populate(rs);   
 **returnnew** ResultSetWrappingSqlRowSet(cachedRowSet);

|  |
| --- |
| } |

|  |
| --- |
| } |

Certain Oracle types require a Connection to obtain the column value from a ResultSet. TIMESTAMPTZ is one of these types. So when rowSet.getTimestamp(colIndex) is called, you will get this exception:

Caused by: java.sql.SQLException: One or more of the authenticating RowSet properties not set at oracle.jdbc.rowset.OracleCachedRowSet.getConnectionInternal(OracleCachedRowSet.java:560) at oracle.jdbc.rowset.OracleCachedRowSet.getTimestamp(OracleCachedRowSet.java:3717) at org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet.getTimestamp

If you dig into this code, you will see that the OracleCachedRowSet needs the password or a JNDI DataSource name to get a Connection. If you prefer the JNDI lookup, just verify that OracleCachedRowSet has DataSourceName set.

So in my Service, I Autowire in the password and declare the output parameter like this:

|  |
| --- |
| **new** SqlOutParameter("cursor\_param\_name", OracleTypes.CURSOR, **new** |

|  |  |
| --- | --- |
| OracleSqlRowSetResultSetExtractor(oraclePassword)) |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 15 |

**Chapter 5: Creating and using beans**

**Section 5.1: Autowiring all beans of a specific type**

If you've got multiple implementations of the same interface, Spring can autowire them all into a collection object.

I'm going to use an example using a Validator pattern1

Foo Class:

|  |
| --- |
| **publicclass** Foo { |

|  |
| --- |
| **private**String name; |

|  |
| --- |
| **private**String emailAddress; |

|  |
| --- |
| **private**String errorMessage; |

|  |
| --- |
| ***/\*\* Getters & Setters omitted \*\*/*** |

|  |
| --- |
| } |

Interface:

|  |
| --- |
| **publicinterface** FooValidator { |

|  |
| --- |
| **public** Foo validate(Foo foo); |

|  |
| --- |
| } |

Name Validator Class:

|  |
| --- |
| @Component(value="FooNameValidator") |

|  |
| --- |
| **publicclass** FooNameValidator **implements** FooValidator { |

|  |
| --- |
| @Override |

|  |
| --- |
| **public** Foo validate(Foo foo){ |

|  |
| --- |
| *//Validation logic goes here.* |

|  |
| --- |
| } |

|  |
| --- |
| } |

Email Validator Class:

|  |
| --- |
| @Component(value="FooEmailValidator") |

|  |
| --- |
| **publicclass** FooEmailValidator **implements** FooValidator { |

|  |
| --- |
| @Override |

|  |
| --- |
| **public** Foo validate(Foo foo){ |

|  |
| --- |
| *//Different validation logic goes here.* |

|  |
| --- |
| } |

|  |
| --- |
| } |

You can now autowire these validators individually or together into a class.

Interface:

|  |
| --- |
| **publicinterface** FooService { |

|  |
| --- |
| **publicvoid** handleFoo(Foo foo); |

|  |
| --- |
| } |

Class:

|  |
| --- |
| @Service |

|  |
| --- |
| **publicclass** FooServiceImpl **implements** FooService { |

|  |
| --- |
| ***/\*\* Autowire all classes implementing FooValidator interface\*\*/*** |

|  |  |
| --- | --- |
| @Autowired |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 16 |

|  |
| --- |
| **private** List<FooValidator> allValidators; |

|  |
| --- |
| @Override |

|  |
| --- |
| **publicvoid** handleFoo(Foo foo){ |

|  |
| --- |
| ***/\*\*You can use all instances from the list\*\*/*** |

|  |
| --- |
| **for**(FooValidator validator : allValidators){ |

|  |
| --- |
| foo = validator.validate(foo); |

|  |
| --- |
| } |

|  |
| --- |
| } |

|  |
| --- |
| } |

It's worth noting that if you have more than one implementation of an interface in the Spring IoC container and

don't specify which one you want to use with the @Qualifier annotation, Spring will throw an exception when

trying to start, because it won't know which instance to use.

1: This is not the right way to do such simple validations. This is a simple example about autowiring. If you want an

idea of a much easier validation method look up how Spring does validation with annotations.

**Section 5.2: Basic annotation autowiring**

Interface:

|  |
| --- |
| **publicinterface** FooService { |

|  |
| --- |
| **publicint** doSomething(); |

|  |
| --- |
| } |

Class:

|  |
| --- |
| @Service |

|  |
| --- |
| **publicclass** FooServiceImpl **implements** FooService { |

|  |
| --- |
| @Override |

|  |
| --- |
| **publicint** doSomething(){ |

|  |
| --- |
| *//Do some stuff here* |

|  |
| --- |
| **return**0; |

|  |
| --- |
| } |

|  |
| --- |
| } |

It should be noted that a class must implement an interface for Spring to be able to autowire this class. There is a

method to allow Spring to autowire stand-alone classes using load time weaving, but that is out of scope for this

example.

You can gain access to this bean in any class that instantiated by the Spring IoC container using the @Autowired

annotation.

Usage:

|  |
| --- |
| @Autowired([required=**true**]) |

The @Autowired annotation will first attempt to autowire by type, and then fall back on bean name in the event of

ambiguity.

This annotation can be applied in several different ways.

Constructor injection:

|  |
| --- |
| **publicclass** BarClass(){ |

|  |  |
| --- | --- |
| **private** FooService fooService |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 17 |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **public** BarClass(FooService fooService){ |

|  |
| --- |
| **this**.fooService= fooService; |

|  |
| --- |
| } |

|  |
| --- |
| } |

Field injection:

|  |
| --- |
| **publicclass** BarClass(){ |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **private** FooService fooService; |

|  |
| --- |
| } |

Setter injection:

|  |
| --- |
| **publicclass** BarClass(){ |

|  |
| --- |
| **private** FooService fooService; |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **publicvoid** setFooService(FooService fooService){ |

|  |
| --- |
| **this**.fooService= fooService; |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Section 5.3: Using FactoryBean for dynamic bean instantiation**

In order to dynamically decide what beans to inject, we can use FactoryBeans. These are classes which implement

the factory method pattern, providing instances of beans for the container. They are recognized by Spring and can

be used transparently, without need to know that the bean comes from a factory. For example:

|  |
| --- |
| **publicclass** ExampleFactoryBean **extends** AbstractFactoryBean<String>{ |

|  |
| --- |
| *// This method determines the type of the bean for autowiring purposes* |

|  |
| --- |
| @Override |

|  |
| --- |
| **public** Class<?> getObjectType(){ |

|  |
| --- |
| **return**String.**class**; |

|  |
| --- |
| } |

|  |
| --- |
| *// this factory method produces the actual bean* |

|  |
| --- |
| @Override |

|  |
| --- |
| **protected**String createInstance()**throws**Exception{ |

|  |
| --- |
| *// The thing you return can be defined dynamically,* |

*// that is read from a file, database, network or just*

*// simply randomly generated if you wish.*

**return**"Something from factory";

}

|  |
| --- |
| } |

Configuration:

|  |
| --- |
| @Configuration |

|  |
| --- |
| **publicclass** ExampleConfig { |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** FactoryBean<String> fromFactory(){ |

|  |
| --- |
| **returnnew** ExampleFactoryBean(); |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 18 |

Getting the bean:

|  |
| --- |
| AbstractApplicationContext context =**new** AnnotationConfigApplicationContext(ExampleConfig.**class**); |

|  |
| --- |
| String exampleString =(String) context.getBean("fromFactory"); |

To get the actual FactoryBean, use the ampersand prefix before the bean's name:

|  |
| --- |
| FactoryBean<String> bean =(FactoryBean<String>) context.getBean("&fromFactory"); |

Please note that you can only use prototype or singleton scopes - to change the scope to prototype override

isSingleton method:

|  |
| --- |
| **publicclass** ExampleFactoryBean **extends** AbstractFactoryBean<String>{ |

|  |
| --- |
| @Override |

|  |
| --- |
| **publicboolean** isSingleton(){ |

|  |
| --- |
| **returnfalse**; |

|  |
| --- |
| } |

|  |
| --- |
| *// other methods omitted for readability reasons* |

|  |
| --- |
| } |

Note that scoping refers to the actual instances being created, not the factory bean itself.

**Section 5.4: Declaring Bean**

To declare a bean, simply annotate a method with the @Bean annotation or annotate a class with the @Component

annotation (annotations @Service, @Repository, @Controller could be used as well).

When JavaConfig encounters such a method, it will execute that method and register the return value as a bean

within a BeanFactory. By default, the bean name will be that of the method name.

We can create bean using one of three ways:

1. **Using Java based Configuration**: In Configuration file we need to declare bean using @bean annotation

|  |
| --- |
| @Configuration |

|  |
| --- |
| **publicclass** AppConfig { |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** TransferService transferService(){ |

|  |
| --- |
| **returnnew** TransferServiceImpl(); |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| 2. | }  **Using XML based configuration**: For XML based configuration we need to create declare bean in application |

configuration XML i.e.

|  |
| --- |
| **<beans>** |

|  |
| --- |
| **<bean**name="transferService"class="com.acme.TransferServiceImpl"**/>** |

|  |  |
| --- | --- |
| 3. | **</beans>**  **Annotation-Driven Component**: For annotation-driven components, we need to add the @Component |

annotation to the class we want to declare as bean.

|  |
| --- |
| @Component("transferService") |

|  |  |
| --- | --- |
| **publicclass** TransferServiceImpl **implements** TransferService { |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 19 |

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

|  |
| --- |
| } |

Now all three beans with name transferService are available in BeanFactory or ApplicationContext.

**Section 5.5: Autowiring specific bean instances with @Qualifier**

If you've got multiple implementations of the same interface, Spring needs to know which one it should autowire

into a class. I'm going to use a Validator pattern in this example.1

Foo Class:

|  |
| --- |
| **publicclass** Foo { |

|  |
| --- |
| **private**String name; |

|  |
| --- |
| **private**String emailAddress; |

|  |
| --- |
| **private**String errorMessage; |

|  |
| --- |
| ***/\*\* Getters & Setters omitted \*\*/*** |

|  |
| --- |
| } |

Interface:

|  |
| --- |
| **publicinterface** FooValidator { |

|  |
| --- |
| **public** Foo validate(Foo foo); |

|  |
| --- |
| } |

Name Validator Class:

|  |
| --- |
| @Component(value="FooNameValidator") |

|  |
| --- |
| **publicclass** FooNameValidator **implements** FooValidator { |

|  |
| --- |
| @Override |

|  |
| --- |
| **public** Foo validate(Foo foo){ |

|  |
| --- |
| *//Validation logic goes here.* |

|  |
| --- |
| } |

|  |
| --- |
| } |

Email Validator Class:

|  |
| --- |
| @Component(value="FooEmailValidator") |

|  |
| --- |
| **publicclass** FooEmailValidator **implements** FooValidator { |

|  |
| --- |
| @Override |

|  |
| --- |
| **public** Foo validate(Foo foo){ |

|  |
| --- |
| *//Different validation logic goes here.* |

|  |
| --- |
| } |

|  |
| --- |
| } |

You can now autowire these validators individually into a class.

Interface:

|  |
| --- |
| **publicinterface** FooService { |

|  |
| --- |
| **publicvoid** handleFoo(Foo foo); |

|  |
| --- |
| } |

Class:

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 20 |

@Service   
**publicclass** FooServiceImpl **implements** FooService {   
 ***/\*\* Autowire validators individually \*\*/***  
 @Autowired   
 */\**  
  *\* Notice how the String value here matches the value*  
  *\* on the @Component annotation? That's how Spring knows which*  *\* instance to autowire.*

*\*/*  
 @Qualifier("FooNameValidator")   
 **private** FooValidator nameValidator;

@Autowired  
 @Qualifier("FooEmailValidator")   
 **private** FooValidator emailValidator;

@Override   
 **publicvoid** handleFoo(Foo foo){   
 ***/\*\*You can use just one instance if you need\*\*/*** foo = nameValidator.validate(foo);   
 }   
}

It's worth noting that if you have more than one implementation of an interface in the Spring IoC container and don't specify which one you want to use with the @Qualifier annotation, Spring will throw an exception when trying to start, because it won't know which instance to use.

1: This is not the right way to do such simple validations. This is a simple example about autowiring. If you want an idea of a much easier validation method look up how Spring does validation with annotations.

**Section 5.6: Autowiring specific instances of classes using generic type parameters**

If you've got an interface with a generic type parameter, Spring can use that to only autowire implementations that implement a type parameter you specify.

Interface:

|  |
| --- |
| **publicinterface** GenericValidator<T>{ |

|  |
| --- |
| **public** T validate(T object); |

|  |
| --- |
| } |

Foo Validator Class:

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** FooValidator **implements** GenericValidator<Foo>{ |

|  |
| --- |
| @Override |

|  |
| --- |
| **public** Foo validate(Foo foo){ |

|  |
| --- |
| *//Logic here to validate foo objects.* |

|  |
| --- |
| } |

|  |
| --- |
| } |

Bar Validator Class:

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** BarValidator **implements** GenericValidator<Bar>{ |

|  |
| --- |
| @Override |

|  |  |
| --- | --- |
| **public** Bar validate(Bar bar){ |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 21 |

|  |
| --- |
| *//Bar validation logic here* |

|  |
| --- |
| } |

|  |
| --- |
| } |

You can now autowire these validators using type parameters to decide which instance to autowire.

Interface:

|  |
| --- |
| **publicinterface** FooService { |

|  |
| --- |
| **publicvoid** handleFoo(Foo foo); |

|  |
| --- |
| } |

Class:

|  |
| --- |
| @Service |

|  |
| --- |
| **publicclass** FooServiceImpl **implements** FooService { |

|  |
| --- |
| ***/\*\* Autowire Foo Validator \*\*/*** |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **private** GenericValidator<Foo> fooValidator; |

|  |
| --- |
| @Override |

|  |
| --- |
| **publicvoid** handleFoo(Foo foo){ |

|  |
| --- |
| foo = fooValidator.validate(foo); |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Section 5.7: Inject prototype-scoped beans into singletons**

The container creates a singleton bean and injects collaborators into it only once. This is not the desired behavior

when a singleton bean has a prototype-scoped collaborator, since the prototype-scoped bean should be injected

every time it is being accessed via accessor.

There are several solutions to this problem:

1. Use lookup method injection

2. Retrieve a prototype-scoped bean via javax.inject.Provider

3. Retrieve a prototype-scoped bean via org.springframework.beans.factory.ObjectFactory (an equivalent

of #2, but with the class that is specific to Spring)

4. Make a singleton bean container aware via implementing ApplicationContextAware interface

Approaches #3 and #4 are generally discouraged, since they strongly tie an app to Spring framework. Thus, they

are not covered in this example.

**Lookup method injection via XML configuration and an abstract method**

Java Classes

|  |
| --- |
| **publicclass**Window{ |

|  |
| --- |
| } |

|  |
| --- |
| **publicabstractclass** WindowGenerator { |

|  |
| --- |
| **public**Window generateWindow(){ |

|  |
| --- |
| Window window = createNewWindow();*// new instance for each call* |

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

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 22 |

|  |
| --- |
| **protectedabstract**Window createNewWindow();*// lookup method* |

|  |
| --- |
| } |

XML

|  |
| --- |
| **<bean**id="window"class="somepackage.Window"scope="prototype"lazy-init="true"**/>** |

|  |
| --- |
| **<bean**id="windowGenerator"class="somepackage.WindowGenerator"**>** |

|  |
| --- |
| **<lookup-method**name="createNewWindow"bean="window"**/>** |

|  |
| --- |
| **</bean>** |

**Lookup method injection via Java configuration and @Component**

Java Classes

**publicclass**Window{

|  |
| --- |
| } |

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** WindowGenerator { |

|  |
| --- |
| **public**Window generateWindow(){ |

Window window = createNewWindow();*// new instance for each call*

...

|  |
| --- |
| } |

|  |
| --- |
| @Lookup |

**protected**Window createNewWindow(){

**thrownew**UnsupportedOperationException();

}

}

Java configuration

|  |
| --- |
| @Configuration |

|  |
| --- |
| @ComponentScan("somepackage")*// package where WindowGenerator is located* |

|  |
| --- |
| **publicclass** MyConfiguration { |

|  |
| --- |
| @Bean |

|  |
| --- |
| @Lazy |

|  |
| --- |
| @Scope(scopeName = ConfigurableBeanFactory.SCOPE\_PROTOTYPE) |

|  |
| --- |
| **public**Window window(){ |

|  |
| --- |
| **returnnew**Window(); |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Manual lookup method injection via Java configuration**

Java Classes

|  |
| --- |
| **publicclass**Window{ |

|  |
| --- |
| } |

|  |
| --- |
| **publicabstractclass** WindowGenerator { |

|  |
| --- |
| **public**Window generateWindow(){ |

|  |
| --- |
| Window window = createNewWindow();*// new instance for each call* |

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

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 23 |

|  |
| --- |
| **protectedabstract**Window createNewWindow();*// lookup method* |

|  |
| --- |
| } |

Java configuration

@Configuration

|  |
| --- |
| **publicclass** MyConfiguration { |

|  |
| --- |
| @Bean |

@Lazy

@Scope(scopeName = ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

**public**Window window(){

**returnnew**Window();

|  |
| --- |
| } |

|  |
| --- |
| @Bean |

**public** WindowGenerator windowGenerator(){

**returnnew** WindowGenerator(){

@Override

**protected**Window createNewWindow(){

**return** window();

}

};

}

}

**Injection of a protoype-scoped bean into singleton via javax.inject.Provider**

Java classes

|  |
| --- |
| **publicclass**Window{ |

|  |
| --- |
| } |

|  |
| --- |
| **publicclass** WindowGenerator { |

|  |
| --- |
| **privatefinal** Provider<Window> windowProvider; |

|  |
| --- |
| **public** WindowGenerator(**final** Provider<Window> windowProvider){ |

|  |
| --- |
| **this**.windowProvider= windowProvider; |

|  |
| --- |
| } |

|  |
| --- |
| **public**Window generateWindow(){ |

|  |
| --- |
| Window window = windowProvider.get();*// new instance for each call* |

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

}

|  |
| --- |
| } |

XML

|  |
| --- |
| **<bean**id="window"class="somepackage.Window"scope="prototype"lazy-init="true"**/>** |

|  |
| --- |
| **<bean**id="windowGenerator"class="somepackage.WindowGenerator"**>** |

|  |
| --- |
| **<constructor-arg>** |

|  |
| --- |
| **<bean**class="org.springframework.beans.factory.config.ProviderCreatingFactoryBean"**>** |

|  |
| --- |
| **<property**name="targetBeanName"value="window"**/>** |

|  |
| --- |
| **</bean>** |

|  |
| --- |
| **</constructor-arg>** |

|  |
| --- |
| **</bean>** |

The same approaches can be used for other scopes as well (e.g. for injection a request-scoped bean into singleton).

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 24 |

**Chapter 6: Bean scopes**

**Section 6.1: Additional scopes in web-aware contexts**

There are several scopes that are available only in a web-aware application context:

 **request** - new bean instance is created per HTTP request

 **session** - new bean instance is created per HTTP session

 **application** - new bean instance is created per ServletContext

 **globalSession** - new bean instance is created per global session in Portlet environment (in Servlet environment global session scope is equal to session scope)

 **websocket** - new bean instance is created per WebSocket session

No additional setup is required to declare and access web-scoped beans in Spring Web MVC environment.

**XML Configuration**

|  |
| --- |
| **<bean**id="myRequestBean"class="OneClass"scope="request"**/>** |

|  |
| --- |
| **<bean**id="mySessionBean"class="AnotherClass"scope="session"**/>** |

|  |
| --- |
| **<bean**id="myApplicationBean"class="YetAnotherClass"scope="application"**/>** |

|  |
| --- |
| **<bean**id="myGlobalSessionBean"class="OneMoreClass"scope="globalSession"**/>** |

**Java Configuration (prior to Spring 4.3)**

@Configuration   
**publicclass** MyConfiguration {

@Bean  
 @Scope(value = WebApplicationContext.SCOPE\_REQUEST, proxyMode = ScopedProxyMode.TARGET\_CLASS) **public** OneClass myRequestBean(){   
 **returnnew** OneClass();   
 }

@Bean  
 @Scope(value = WebApplicationContext.SCOPE\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS) **public** AnotherClass mySessionBean(){   
 **returnnew** AnotherClass();   
 }

@Bean  
 @Scope(value = WebApplicationContext.SCOPE\_APPLICATION, proxyMode = ScopedProxyMode.TARGET\_CLASS)   
 **public** YetAnotherClass myApplicationBean(){   
 **returnnew** YetAnotherClass();   
 }

@Bean  
 @Scope(value = WebApplicationContext.SCOPE\_GLOBAL\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)   
 **public** OneMoreClass myGlobalSessionBean(){   
 **returnnew** OneMoreClass();   
 }   
}

**Java Configuration (after Spring 4.3)**

|  |
| --- |
| @Configuration |

|  |  |
| --- | --- |
| **publicclass** MyConfiguration { |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 25 |

@Bean  
 @RequestScope   
 **public** OneClass myRequestBean(){   
 **returnnew** OneClass();

|  |
| --- |
| } |

|  |
| --- |
| @Bean |

@SessionScope   
 **public** AnotherClass mySessionBean(){   
 **returnnew** AnotherClass();

|  |
| --- |
| } |

|  |
| --- |
| @Bean |

@ApplicationScope   
 **public** YetAnotherClass myApplicationBean(){   
 **returnnew** YetAnotherClass();   
 }   
}

**Annotation-Driven Components**

@Component   
@RequestScope   
**publicclass** OneClass {  
 ...

}

@Component   
@SessionScope   
**publicclass** AnotherClass {  
 ...

}

@Component   
@ApplicationScope   
**publicclass** YetAnotherClass {  
 ...

}

@Component   
@Scope(scopeName = WebApplicationContext.SCOPE\_GLOBAL\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)   
**publicclass** OneMoreClass {  
 ...

}

@Component   
@Scope(scopeName ="websocket", proxyMode = ScopedProxyMode.TARGET\_CLASS) **publicclass** AndOneMoreClass {  
 ...

}

**Section 6.2: Prototype scope**

A prototype-scoped bean is not pre-created on Spring container startup. Instead, a new fresh instance will be created every time a request to retrieve this bean is sent to the container. This scope is recommended for stateful objects, since its state won't be shared by other components.

In order to define a prototype-scoped bean, we need to add the @Scope annotation, specifying the type of scope we want.

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 26 |

Given the following MyBean class:

**publicclass** MyBean {

**privatestaticfinal** Logger LOGGER = LoggerFactory.getLogger(MyBean.**class**);

|  |
| --- |
| **private**String property; |

|  |
| --- |
| **public** MyBean(String property){ |

**this**.property= property;

LOGGER.info("Initializing {} bean...", property);

|  |
| --- |
| } |

|  |
| --- |
| **public**String getProperty(){ |

**returnthis**.property;

|  |
| --- |
| } |

|  |
| --- |
| **publicvoid** setProperty(String property){ |

**this**.property= property;

}

}

We define a bean definition, stating its scope as prototype:

|  |
| --- |
| @Configuration |

|  |
| --- |
| **publicclass** PrototypeConfiguration { |

|  |
| --- |
| @Bean |

|  |
| --- |
| @Scope("prototype") |

|  |
| --- |
| **public** MyBean prototypeBean(){ |

|  |
| --- |
| **returnnew** MyBean("prototype"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

In order to see how it works, we retrieve the bean from the Spring container and set a different value for its

property field. Next, we will again retrieve the bean from the container and look up its value:

|  |
| --- |
| MyBean prototypeBean1 = context.getBean("prototypeBean", MyBean.**class**); |

|  |
| --- |
| prototypeBean1.setProperty("changed property"); |

|  |
| --- |
| MyBean prototypeBean2 = context.getBean("prototypeBean", MyBean.**class**); |

|  |
| --- |
| logger.info("Prototype bean 1 property: "+ prototypeBean1.getProperty()); |

|  |
| --- |
| logger.info("Prototype bean 2 property: "+ prototypeBean2.getProperty()); |

Looking at the following result, we can see how a new instance has been created on each bean request:

|  |
| --- |
| Initializing prototype bean... |

|  |
| --- |
| Initializing prototype bean... |

|  |
| --- |
| Prototype bean 1 property: changed property |

|  |
| --- |
| Prototype bean 2 property: prototype |

A common mistake is to assume that the bean is recreated per invocation or per thread, this is **NOT** the case.

Instead an instance is created PER INJECTION (or retrieval from the context). If a Prototype scoped bean is only ever

injected into a single singleton bean, there will only ever be one instance of that Prototype scoped bean.

Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures,

decorates and otherwise assembles a prototype object, hands it to the client and then has no further knowledge of

that prototype instance.

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 27 |

**Section 6.3: Singleton scope**

If a bean is defined with singleton scope, there will only be one single object instance initialized in the Spring container. All requests to this bean will return the same shared instance. This is the default scope when defining a bean.

Given the following MyBean class:

**publicclass** MyBean {   
 **privatestaticfinal** Logger LOGGER = LoggerFactory.getLogger(MyBean.**class**);

|  |
| --- |
| **private**String property; |

|  |
| --- |
| **public** MyBean(String property){ |

**this**.property= property;  
 LOGGER.info("Initializing {} bean...", property);

|  |
| --- |
| } |

|  |
| --- |
| **public**String getProperty(){ |

**returnthis**.property;

|  |
| --- |
| } |

|  |
| --- |
| **publicvoid** setProperty(String property){ |

**this**.property= property;   
 }   
}

We can define a singleton bean with the @Bean annotation:

|  |
| --- |
| @Configuration |

|  |
| --- |
| **publicclass** SingletonConfiguration { |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** MyBean singletonBean(){ |

|  |
| --- |
| **returnnew** MyBean("singleton"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

The following example retrieves the same bean twice from the Spring context:

|  |
| --- |
| MyBean singletonBean1 = context.getBean("singletonBean", MyBean.**class**); |

|  |
| --- |
| singletonBean1.setProperty("changed property"); |

|  |
| --- |
| MyBean singletonBean2 = context.getBean("singletonBean", MyBean.**class**); |

When logging the singletonBean2 property, the message *"changed property"* will be shown, since we just retrieved the same shared instance.

Since the instance is shared among different components, it is recommended to define singleton scope for stateless objects.

**Lazy singleton beans**

By default, singleton beans are pre-instantiated. Hence, the shared object instance will be created when the Spring container is created. If we start the application, the *"Initializing singleton bean..."* message will be shown.

If we don't want the bean to be pre-instantiated, we can add the @Lazy annotation to the bean definition. This will prevent the bean from being created until it is first requested.

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 28 |

|  |
| --- |
| @Bean |

|  |
| --- |
| @Lazy |

|  |
| --- |
| **public** MyBean lazySingletonBean(){ |

|  |
| --- |
| **returnnew** MyBean("lazy singleton"); |

|  |
| --- |
| } |

Now, if we start the Spring container, no *"Initializing lazy singleton bean..."* message will appear. The bean won't be

created until it is requested for the first time:

|  |
| --- |
| logger.info("Retrieving lazy singleton bean..."); |

|  |
| --- |
| context.getBean("lazySingletonBean"); |

If we run the application with both singleton and lazy singleton beans defined, It will produce the following

messages:

|  |
| --- |
| Initializing singleton bean... |

|  |
| --- |
| Retrieving lazy singleton bean... |

|  |  |
| --- | --- |
| Initializing lazy singleton bean... |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 29 |

**Chapter 7: Conditional bean registration in Spring**

**Section 7.1: Register beans only when a property or value is specified**

A spring bean can be configured such that it will register *only* if it has a particular value *or* a specified property is

met. To do so, implement [Condition.matches](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Condition.html#matches-org.springframework.context.annotation.ConditionContext-org.springframework.core.type.AnnotatedTypeMetadata-) to check the property/value:

|  |
| --- |
| **publicclass** PropertyCondition **implements** Condition { |

|  |
| --- |
| @Override |

|  |
| --- |
| **publicboolean** matches(ConditionContext context, AnnotatedTypeMetadata metadata){ |

|  |
| --- |
| **return** context.getEnvironment().getProperty("propertyName")!=**null**; |

|  |
| --- |
| *// optionally check the property value* |

|  |
| --- |
| } |

|  |
| --- |
| } |

In Java config, use the above implementation as a condition to register the bean. Note the use of [@Conditional](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Conditional.html)

annotation.

|  |
| --- |
| @Configuration |

|  |
| --- |
| **publicclass** MyAppConfig { |

|  |
| --- |
| @Bean |

|  |
| --- |
| @Conditional(PropertyCondition.**class**) |

|  |
| --- |
| **public** MyBean myBean(){ |

|  |
| --- |
| **returnnew** MyBean(); |

|  |
| --- |
| } |

|  |
| --- |
| } |

In PropertyCondition, any number of conditions can be evaluated. However it is advised to separate the

implementation for each condition to keep them loosely coupled. For example:

|  |
| --- |
| @Configuration |

|  |
| --- |
| **publicclass** MyAppConfig { |

|  |
| --- |
| @Bean |

|  |
| --- |
| @Conditional({PropertyCondition.**class**, SomeOtherCondition.**class**}) |

|  |
| --- |
| **public** MyBean myBean(){ |

|  |
| --- |
| **returnnew** MyBean(); |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Section 7.2: Condition annotations**

Except main @conditional annotation there are set of similar annotation to be used for different cases.

**Class conditions**

The @ConditionalOnClass and @ConditionalOnMissingClass annotations allows configuration to be included

based on the presence or absence of specific classes.

E.g. when OObjectDatabaseTx.**class** is added to dependencies and there is no OrientWebConfigurer bean we

create the configurer.

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 30 |

|  |
| --- |
| @Bean |

|  |
| --- |
| @ConditionalOnWebApplication |

|  |
| --- |
| @ConditionalOnClass(OObjectDatabaseTx.**class**) |

|  |
| --- |
| @ConditionalOnMissingBean(OrientWebConfigurer.**class**) |

|  |
| --- |
| **public** OrientWebConfigurer orientWebConfigurer(){ |

|  |
| --- |
| **returnnew** OrientWebConfigurer(); |

|  |
| --- |
| } |

**Bean conditions**

The @ConditionalOnBean and @ConditionalOnMissingBean annotations allow a bean to be included based on the

presence or absence of specific beans. You can use the value attribute to specify beans by type, or name to specify

beans by name. The search attribute allows you to limit the ApplicationContext hierarchy that should be

considered when searching for beans.

See the example above when we check whether there is no defined bean.

**Property conditions**

The @ConditionalOnProperty annotation allows configuration to be included based on a Spring Environment

property. Use the prefix and name attributes to specify the property that should be checked. By default any

property that exists and is not equal to **false** will be matched. You can also create more advanced checks using the

havingValue and matchIfMissing attributes.

|  |
| --- |
| @ConditionalOnProperty(value='somebean.enabled', matchIfMissing =**true**, havingValue="yes") |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** SomeBean someBean(){ |

|  |
| --- |
| } |

**Resource conditions**

The @ConditionalOnResource annotation allows configuration to be included only when a specific resource is

present.

|  |
| --- |
| @ConditionalOnResource(resources ="classpath:init-db.sql") |

**Web application conditions**

The @ConditionalOnWebApplication and @ConditionalOnNotWebApplication annotations allow configuration to

be included depending on whether the application is a 'web application'.

|  |
| --- |
| @Configuration |

|  |
| --- |
| @ConditionalOnWebApplication |

|  |
| --- |
| **publicclass** MyWebMvcAutoConfiguration {...} |

**SpEL expression conditions**

The @ConditionalOnExpression annotation allows configuration to be included based on the result of a SpEL

expression.

|  |  |  |
| --- | --- | --- |
| |  | | --- | | @ConditionalOnExpression("${rest.security.enabled}==false") | |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 31 |

**Chapter 8: Spring JSR 303 Bean Validation**

Spring has JSR303 bean validation support. We can use this to do input bean validation. Separate validation logic

from business logic using JSR303.

**Section 8.1: @Valid usage to validate nested POJOs**

Suppose we have a POJO class User we need to validate.

|  |
| --- |
| **publicclass** User { |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| @Size(min=5) |

|  |
| --- |
| @Email |

|  |
| --- |
| **private**String email; |

|  |
| --- |
| } |

and a controller method to validate the user instance

|  |
| --- |
| **public**String registerUser(@Valid User user, BindingResult result); |

Let's extend the User with a nested POJO Address we also need to validate.

|  |
| --- |
| **publicclass** Address { |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| @Size(min=2, max=3) |

|  |
| --- |
| **private**String countryCode; |

|  |
| --- |
| } |

Just add @Valid annotation on address field to run validation of nested POJOs.

|  |
| --- |
| **publicclass** User { |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| @Size(min=5) |

|  |
| --- |
| @Email |

|  |
| --- |
| **private**String email; |

|  |
| --- |
| @Valid |

|  |
| --- |
| **private** Address address; |

|  |
| --- |
| } |

**Section 8.2: Spring JSR 303 Validation - Customize error messages**

Suppose we have a simple class with validation annotations

|  |
| --- |
| **publicclass** UserDTO { |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| **private**String name; |

|  |
| --- |
| @Min(18) |

|  |
| --- |
| **privateint** age; |

|  |  |
| --- | --- |
| *//getters/setters* |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 32 |

|  |
| --- |
| } |

A controller to check the UserDTO validity.

|  |
| --- |
| @RestController |

|  |
| --- |
| **publicclass** ValidationController { |

|  |
| --- |
| @RequestMapping(value ="/validate", method = RequestMethod.POST) |

|  |
| --- |
| **public** ResponseEntity<String> check(@Valid @RequestBody UserDTO userDTO, |

|  |
| --- |
| BindingResult bindingResult){ |

|  |
| --- |
| **returnnew** ResponseEntity<>("ok" , HttpStatus.OK); |

|  |
| --- |
| } |

|  |
| --- |
| } |

And a test.

@Test

**publicvoid** testValid()**throws**Exception{

TestRestTemplate template =**new** TestRestTemplate();

String url = base + contextPath +"/validate";

Map<String, Object> params =**new** HashMap<>();

params.put("name", "");

|  |
| --- |
| params.put("age", "10"); |

|  |
| --- |
| MultiValueMap<String, String> headers =**new** LinkedMultiValueMap<>(); |

|  |
| --- |
| headers.add("Content-Type", "application/json"); |

|  |
| --- |
| HttpEntity<Map<String, Object>> request =**new** HttpEntity<>(params, headers); |

|  |
| --- |
| String res = template.postForObject(url, request, String.**class**); |

|  |
| --- |
| assertThat(res, equalTo("ok")); |

}

Both name and age are invalid so in the BindingResult we have two validation errors. Each has array of codes.

Codes for Min check

|  |
| --- |
| 0="Min.userDTO.age" |

|  |
| --- |
| 1="Min.age" |

|  |
| --- |
| 2="Min.int" |

|  |
| --- |
| 3="Min" |

And for NotEmpty check

|  |
| --- |
| 0="NotEmpty.userDTO.name" |

|  |
| --- |
| 1="NotEmpty.name" |

|  |
| --- |
| 2="NotEmpty.java.lang.String" |

|  |
| --- |
| 3="NotEmpty" |

Let's add a custom.properties file to substitute default messages.

|  |
| --- |
| @SpringBootApplication |

|  |
| --- |
| @Configuration |

|  |
| --- |
| **publicclass** DemoApplication { |

|  |
| --- |
| @Bean(name ="messageSource") |

|  |
| --- |
| **public** MessageSource messageSource(){ |

|  |
| --- |
| ReloadableResourceBundleMessageSource bean =**new** ReloadableResourceBundleMessageSource(); |

|  |  |
| --- | --- |
| bean.setBasename("classpath:custom"); |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 33 |

bean.setDefaultEncoding("UTF-8");   
 **return** bean;

|  |
| --- |
| } |

|  |
| --- |
| @Bean(name ="validator") |

**public** LocalValidatorFactoryBean validator(){  
 LocalValidatorFactoryBean bean =**new** LocalValidatorFactoryBean(); bean.setValidationMessageSource(messageSource());   
 **return** bean;

|  |
| --- |
| } |

|  |
| --- |
| **publicstaticvoid** main(String[] args){ |

SpringApplication.run(DemoApplication.**class**, args); }   
}

If we add to the custom.properties file the line

|  |
| --- |
| NotEmpty=The field must not be empty! |

The new value is shown for the error. To resolve message validator looks through the codes starting from the beginning to find proper messages.

Thus when we define NotEmpty key in the .properties file for all cases where the @NotEmpty annotation is used our message is applied.

If we define a message

|  |
| --- |
| Min.**int**=Some custom message here. |

All annotations where we app min check to integer values use the newly defined message.

The same logic could be applied if we need to localize the validation error messages.

**Section 8.3: JSR303 Annotation based validations in Springs examples**   
Add any JSR 303 implementation to your classpath. Popular one used is Hibernate validator from Hibernate.

|  |
| --- |
| **<dependency>** |

|  |
| --- |
| **<groupId>**org.hibernate**</groupId>** |

|  |
| --- |
| **<artifactId>**hibernate-validator**</artifactId>** |

|  |
| --- |
| **<version>**4.2.0.Final**</version>** |

|  |
| --- |
| **</dependency>** |

Lets say the there is a rest api to create user in the system

|  |
| --- |
| @RequestMapping(value="/registeruser", method=RequestMethod.POST) |

|  |
| --- |
| **public**String registerUser(User user); |

The input json sample would look like as below

|  |
| --- |
| {"username":"abc@abc.com","password":"password1","password2":"password1"} |

User.java

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 34 |

|  |
| --- |
| **publicclass** User { |

|  |
| --- |
| **private**String username; |

|  |
| --- |
| **private**String password; |

|  |
| --- |
| **private**String password2; |

|  |
| --- |
| getXXX and setXXX |

|  |
| --- |
| } |

We can define JSR 303 validations on User Class as below.

|  |
| --- |
| **publicclass** User { |

|  |
| --- |
| @NotEmpty |

@Size(min=5)

@Email

|  |
| --- |
| **private**String username; |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| **private**String password; |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| **private**String password2; |

|  |
| --- |
| } |

We may also need to have a business validator like password and password2(confirm password) are same, for this

we can add a custom validator as below. Write a custom annotation for annotating the data field.

|  |
| --- |
| @Target({ ElementType.FIELD}) |

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME) |

|  |
| --- |
| @Constraint(validatedBy = PasswordValidator.**class**) |

|  |
| --- |
| **public** @**interface** GoodPassword { |

|  |
| --- |
| String message()**default**"Passwords won't match."; |

|  |
| --- |
| Class<?>[] groups()**default**{}; |

|  |
| --- |
| Class<?**extends** Payload>[] payload()**default**{}; |

|  |
| --- |
| } |

Write a Validator class for applying Validation logic.

|  |
| --- |
| **publicclass** PastValidator **implements** ConstraintValidator<GoodPassword, User>{ |

|  |
| --- |
| @Override |

|  |
| --- |
| **publicvoid** initialize(GoodPassword annotation){} |

|  |
| --- |
| @Override |

|  |
| --- |
| **publicboolean** isValid(User user, ConstraintValidatorContext context){ |

|  |
| --- |
| **return** user.getPassword().equals(user.getPassword2()); |

|  |
| --- |
| } |

|  |
| --- |
| } |

Adding this validation to User Class

|  |
| --- |
| @GoodPassword |

|  |
| --- |
| **publicclass** User { |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| @Size(min=5) |

|  |  |
| --- | --- |
| @Email |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 35 |

|  |
| --- |
| **private**String username; |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| **private**String password; |

|  |
| --- |
| @NotEmpty |

|  |
| --- |
| **private**String password2; |

|  |
| --- |
| } |

@Valid triggers validation in Spring. BindingResult is an object injected by spring which has list of errors after validation.

|  |
| --- |
| **public**String registerUser(@Valid User user, BindingResult result); |

JSR 303 annotation has message attributes on them which can be used for providing custom messages.

@GoodPassword

|  |
| --- |
| **publicclass** User { |

|  |
| --- |
| @NotEmpty(message="Username Cant be empty") |

@Size(min=5, message="Username cant be les than 5 chars") @Email(message="Should be in email format")

|  |
| --- |
| **private**String username; |

|  |
| --- |
| @NotEmpty(message="Password cant be empty") |

|  |
| --- |
| **private**String password; |

|  |
| --- |
| @NotEmpty(message="Password2 cant be empty") |

|  |
| --- |
| **private**String password2; |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 36 |

**Chapter 9: ApplicationContext Configuration**

**Section 9.1: Autowiring**

Autowiring is done using a *sterotype* annotation to specify what classes are going to be beans in the   
ApplicationContext, and using the Autowired and Value annotations to specify bean dependencies. The unique part of autowiring is that there is no external ApplicationContext definition, as it is all done within the classes that are the beans themselves.

@Component*// The annotation that specifies to include this as a bean*  *// in the ApplicationContext*   
**class**Book{

@Autowired *// The annotation that wires the below defined Author*  *// instance into this bean*  
 Author author;

|  |
| --- |
| String title ="It"; |

Author getAuthor(){**return** author;}   
 String getTitle(){**return** title;}   
}

@Component*// The annotation that specifies to include*  *// this as a bean in the ApplicationContext* **class** Author {   
 String firstName ="Steven";   
 String lastName ="King";

String getFirstName(){**return** firstName;}   
 String getLastName(){**return** lastName;}   
}

**Section 9.2: Bootstrapping the ApplicationContext**

**Java Config**

The configuration class needs only to be a class that is on the classpath of your application and visible to your applications main class.

|  |
| --- |
| **class** MyApp { |

|  |
| --- |
| **publicstaticvoid** main(String[] args)**throws**Exception{ |

|  |
| --- |
| AnnotationConfigApplicationContext appContext = |

|  |
| --- |
| **new** AnnotationConfigApplicationContext(MyConfig.**class**); |

|  |
| --- |
| *// ready to retrieve beans from appContext, such as myObject.* |

}

|  |
| --- |
| } |

|  |
| --- |
| @Configuration |

**class** MyConfig {  
 @Bean  
 MyObject myObject(){   
 *// ...configure myObject...*

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 37 |

|  |
| --- |
| *// ...define more beans...* |

|  |
| --- |
| } |

**Xml Config**

The configuration xml file needs only be on the classpath of your application.

**class** MyApp {   
 **publicstaticvoid** main(String[] args)**throws**Exception{ ClassPathXmlApplicationContext appContext =

|  |
| --- |
| **new** ClassPathXmlApplicationContext("applicationContext.xml"); |

|  |
| --- |
| *// ready to retrieve beans from appContext, such as myObject.* |

}

|  |
| --- |
| } |

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

<!-- applicationContext.xml-->   
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd"> <bean id="myObject"**class**="com.example.MyObject">   
 <!-- ...configure myObject... -->

|  |
| --- |
| </bean> |

|  |
| --- |
| <!-- ...define more beans... --> |

</beans>

**Autowiring**

Autowiring needs to know which base packages to scan for annotated beans (@Component). This is specified via the #scan(String...) method.

|  |
| --- |
| **class** MyApp { |

|  |
| --- |
| **publicstaticvoid** main(String[] args)**throws**Exception{ |

|  |
| --- |
| AnnotationConfigApplicationContext appContext = |

|  |
| --- |
| **new** AnnotationConfigApplicationContext(); |

|  |
| --- |
| appContext.scan("com.example"); |

|  |
| --- |
| appContext.refresh(); |

|  |
| --- |
| *// ready to retrieve beans from appContext, such as myObject.* |

|  |
| --- |
| } |

|  |
| --- |
| } |

|  |
| --- |
| *// assume this class is in the com.example package.* |

@Component   
**class** MyObject {   
 *// ...myObject definition...*

|  |
| --- |
| } |

**Section 9.3: Java Configuration**

Java configuration is typically done by applying the @Configuration annotation to a class to suggest that a class contains bean definitions. The bean definitions are specified by applying the @Bean annotation to a method that returns an object.

|  |  |  |
| --- | --- | --- |
| |  | | --- | | @Configuration *// This annotation tells the ApplicationContext that this class* | |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 38 |

*// contains bean definitions.*

**class** AppConfig {   
 ***/\*\****  
  ***\* An Author created with the default constructor***  
  ***\* setting no properties***  
  ***\*/***  
 @Bean *// This annotation marks a method that defines a bean* Author author1(){   
 **returnnew** Author();   
 }

***/\*\****  
 ***\* An Author created with the constructor that initializes the \* name fields***  
 ***\*/***  
 @Bean  
 Author author2(){   
 **returnnew** Author("Steven", "King");   
 }

***/\*\****  
 ***\* An Author created with the default constructor, but \* then uses the property setters to specify name fields \*/***  
 @Bean  
 Author author3(){  
 Author author =**new** Author();  
 author.setFirstName("George");  
 author.setLastName("Martin");   
 **return** author;   
 }

***/\*\****  
 ***\* A Book created referring to author2 (created above) via \* a constructor argument. The dependency is fulfilled by \* invoking the method as plain Java.***

***\*/***  
 @Bean   
 Book book1(){   
 **returnnew**Book(author2(), "It");   
 }

***/\*\****  
 ***\* A Book created referring to author3 (created above) via \* a property setter. The dependency is fulfilled by \* invoking the method as plain Java.***

***\*/***  
 @Bean   
 Book book2(){   
 Book book =**new**Book();  
 book.setAuthor(author3());  
 book.setTitle("A Game of Thrones");   
 **return** book;   
 }   
}

|  |
| --- |
| *// The classes that are being initialized and wired above...* |

|  |
| --- |
| **class**Book{*// assume package org.springframework.example* |

|  |
| --- |
| Author author; |

|  |  |
| --- | --- |
| String title; |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 39 |

Book(){}*// default constructor*   
Book(Author author, String title){   
 **this**.author= author;   
 **this**.title= title;   
}

Author getAuthor(){**return** author;}   
 String getTitle(){**return** title;}

**void** setAuthor(Author author){   
 **this**.author= author;   
}

**void** setTitle(String title){   
 **this**.title= title;   
 }   
}

**class** Author {*// assume package org.springframework.example* String firstName;   
String lastName;

Author(){}*// default constructor*  
 Author(String firstName, String lastName){   
 **this**.firstName= firstName;   
 **this**.lastName= lastName;   
 }

String getFirstName(){**return** firstName;}   
String getLastName(){**return** lastName;}

**void** setFirstName(String firstName){   
 **this**.firstName= firstName;   
}

**void** setLastName(String lastName){   
 **this**.lastName= lastName;   
 }   
}

**Section 9.4: Xml Configuration**

Xml configuration is typically done by defining beans within an xml file, using Spring's specific beans schema. Under the root beans element, typical bean definition would be done using the bean subelement.

**<?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.xsd"**>**  *<!-- An Author created with the default constructor*  
  *setting no properties -->*

|  |
| --- |
| **<bean**id="author1"class="org.springframework.example.Author"**/>** |

|  |
| --- |
| *<!-- An Author created with the constructor that initializes the* |

*name fields -->*   
**<bean**id="author2"class="org.springframework.example.Author"**>**  **<constructor-arg**index="0"value="Steven"**/>**   
 **<constructor-arg**index="1"value="King"**/>**

|  |  |
| --- | --- |
| **</bean>** |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 40 |

*<!-- An Author created with the default constructor, but*   
  *then uses the property setters to specify name fields -->* **<bean**id="author3"class="org.springframework.example.Author"**>**  **<property**name="firstName"value="George"**/>**   
 **<property**name="lastName"value="Martin"**/>**

|  |
| --- |
| **</bean>** |

|  |
| --- |
| *<!-- A Book created referring to author2 (created above) via* |

*a constructor argument -->*   
**<bean**id="book1"class="org.springframework.example.Book"**>**  **<constructor-arg**index="0"ref="author2"**/>**   
 **<constructor-arg**index="1"value="It"**/>**

|  |
| --- |
| **</bean>** |

|  |
| --- |
| *<!-- A Book created referring to author3 (created above) via* |

*a property setter -->*   
 **<bean**id="book1"class="org.springframework.example.Book"**>**  **<property**name="author"ref="author3"**/>**   
 **<property**name="title"value="A Game of Thrones"**/>**  **</bean>**   
**</beans>**

*// The classes that are being initialized and wired above...* **class**Book{*// assume package org.springframework.example* Author author;   
 String title;

Book(){}*// default constructor*   
Book(Author author, String title){   
 **this**.author= author;   
 **this**.title= title;   
}

Author getAuthor(){**return** author;}   
 String getTitle(){**return** title;}

**void** setAuthor(Author author){   
 **this**.author= author;   
}

**void** setTitle(String title){   
 **this**.title= title;   
 }   
}

**class** Author {*// assume package org.springframework.example* String firstName;   
String lastName;

Author(){}*// default constructor*  
 Author(String firstName, String lastName){   
 **this**.firstName= firstName;   
 **this**.lastName= lastName;   
 }

String getFirstName(){**return** firstName;}   
String getLastName(){**return** lastName;}

**void** setFirstName(String firstName){   
 **this**.firstName= firstName;

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 41 |

|  |
| --- |
| **void** setLastName(String lastName){ |

|  |
| --- |
| **this**.lastName= lastName; |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 42 |

**Chapter 10: RestTemplate**

**Section 10.1: Downloading a Large File**

The getForObject and getForEntity methods of RestTemplate load the entire response in memory. This is not

suitable for downloading large files since it can cause out of memory exceptions. This example shows how to

stream the response of a GET request.

|  |
| --- |
| RestTemplate restTemplate *// = ...;* |

|  |
| --- |
| *// Optional Accept header* |

RequestCallback requestCallback = request -> request.getHeaders()

|  |
| --- |
| .setAccept(Arrays.asList(MediaType.APPLICATION\_OCTET\_STREAM, MediaType.ALL)); |

|  |
| --- |
| *// Streams the response instead of loading it all in memory* |

ResponseExtractor<Void> responseExtractor = response ->{

*// Here I write the response to a file but do what you like*

Path path = Paths.get("some/path");

Files.copy(response.getBody(), path);

**returnnull**;

};

restTemplate.execute(URI.create("www.something.com"), HttpMethod.GET, requestCallback,

responseExtractor);

Note that you cannot simply return the InputStream from the extractor, because by the time the execute method

returns, the underlying connection and stream are already closed.

**Section 10.2: Setting headers on Spring RestTemplate request**

The exchange methods of RestTemplate allows you specify a HttpEntity that will be written to the request when

execute the method. You can add headers (such user agent, referrer...) to this entity:

|  |
| --- |
| **publicvoid** testHeader(**final** RestTemplate restTemplate){ |

|  |
| --- |
| *//Set the headers you need send* |

|  |
| --- |
| **final** HttpHeaders headers =**new** HttpHeaders(); |

|  |
| --- |
| headers.set("User-Agent", "eltabo"); |

|  |
| --- |
| *//Create a new HttpEntity* |

|  |
| --- |
| **final** HttpEntity<String> entity =**new** HttpEntity<String>(headers); |

|  |
| --- |
| *//Execute the method writing your HttpEntity to the request* |

|  |
| --- |
| ResponseEntity<Map> response = restTemplate.exchange("https://httpbin.org/user-agent", |

|  |
| --- |
| HttpMethod.GET, entity, Map.**class**); |

|  |
| --- |
| System.out.println(response.getBody()); |

|  |
| --- |
| } |

Also you can add an interceptor to your RestTemplate if you need to add the same headers to multiple requests:

|  |
| --- |
| **publicvoid** testHeader2(**final** RestTemplate restTemplate){ |

|  |
| --- |
| *//Add a ClientHttpRequestInterceptor to the RestTemplate* |

|  |
| --- |
| restTemplate.getInterceptors().add(**new** ClientHttpRequestInterceptor(){ |

|  |
| --- |
| @Override |

|  |
| --- |
| **public** ClientHttpResponse intercept(HttpRequest request, **byte**[] body, |

|  |
| --- |
| ClientHttpRequestExecution execution)**throws**IOException{ |

|  |
| --- |
| request.getHeaders().set("User-Agent", "eltabo");*//Set the header for each request* |

|  |
| --- |
| **return** execution.execute(request, body); |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 43 |

|  |
| --- |
| }); |

|  |
| --- |
| ResponseEntity<Map> response = restTemplate.getForEntity("https://httpbin.org/user-agent", |

|  |
| --- |
| Map.**class**); |

|  |
| --- |
| System.out.println(response.getBody()); |

|  |
| --- |
| ResponseEntity<Map> response2 = restTemplate.getForEntity("https://httpbin.org/headers", |

|  |
| --- |
| Map.**class**); |

|  |
| --- |
| System.out.println(response2.getBody()); |

|  |
| --- |
| } |

**Section 10.3: Generics results from Spring RestTemplate**

To let RestTemplate understand generic of returned content we need to define result type reference.

org.springframework.core.ParameterizedTypeReference has been introduced since 3.2

|  |
| --- |
| Wrapper<Model> response = restClient.exchange(url, |

|  |
| --- |
| HttpMethod.GET, |

|  |
| --- |
| **null**, |

|  |
| --- |
| **new** ParameterizedTypeReference<Wrapper<Model>>(){}).getBody(); |

Could be useful to get e.g. List<User> from a controller.

**Section 10.4: Using Preemptive Basic Authentication with RestTemplate and HttpClient**

Preemptive basic authentication is the practice of sending http basic authentication credentials (username and

password) *before* a server replies with a 401 response asking for them. This can save a request round trip when

consuming REST apis which are known to require basic authentication.

As described in the [Spring documentation](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#rest-resttemplate), [Apache HttpClient](https://hc.apache.org/httpcomponents-client-ga/) may be used as the underlying implementation to

create HTTP requests by using the HttpComponentsClientHttpRequestFactory. HttpClient can be configured to do

[preemptive basic authentication](https://hc.apache.org/httpcomponents-client-ga/tutorial/html/authentication.html#d5e717).

The following class extends HttpComponentsClientHttpRequestFactory to provide preemptive basic

authentication.

***/\*\****

***\* {@link HttpComponentsClientHttpRequestFactory} with preemptive basic***

***\* authentication to avoid the unnecessary first 401 response asking for***

***\* credentials.***

***\* <p>***

***\* Only preemptively sends the given credentials to the given host and***

***\* optionally to its subdomains. Matching subdomains can be useful for APIs***

***\* using multiple subdomains which are not always known in advance.***

***\* <p>***

***\* Other configurations of the {@link HttpClient} are not modified (e.g. the***

***\* default credentials provider).***

***\*/***

**publicclass** PreAuthHttpComponentsClientHttpRequestFactory **extends**

|  |
| --- |
| HttpComponentsClientHttpRequestFactory { |

|  |
| --- |
| **private**String hostName; |

**privateboolean** matchSubDomains;

|  |
| --- |
| **private** Credentials credentials; |

|  |  |
| --- | --- |
| ***/\*\**** |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 44 |

***\* @param httpClient***  
 ***\* client***  
 ***\* @param hostName***  
 ***\* host name***  
 ***\* @param matchSubDomains***  
 ***\* whether to match the host's subdomains***  
 ***\* @param userName***  
 ***\* basic authentication user name***  
 ***\* @param password***  
 ***\* basic authentication password***  
 ***\*/***   
**public** PreAuthHttpComponentsClientHttpRequestFactory(HttpClient httpClient, String hostName, **boolean** matchSubDomains, String userName, String password){   
 **super**(httpClient);   
 **this**.hostName= hostName;   
 **this**.matchSubDomains= matchSubDomains;  
 credentials =**new** UsernamePasswordCredentials(userName, password);   
}

@Override   
 **protected** HttpContext createHttpContext(HttpMethod httpMethod, URI uri){ *// Add AuthCache to the execution context*  
 HttpClientContext context = HttpClientContext.create();  
 context.setCredentialsProvider(**new** PreAuthCredentialsProvider()); context.setAuthCache(**new** PreAuthAuthCache());   
 **return** context;   
 }

***/\*\****  
 ***\* @param host***  
 ***\* host name***  
 ***\* @return whether the configured credentials should be used for the given***  
 ***\* host***  
 ***\*/***   
**protectedboolean** hostNameMatches(String host){   
 **return** host.equals(hostName)||(matchSubDomains && host.endsWith("."+ hostName)); }

**privateclass** PreAuthCredentialsProvider **extends** BasicCredentialsProvider { @Override   
 **public** Credentials getCredentials(AuthScope authscope){   
 **if**(hostNameMatches(authscope.getHost())){   
 *// Simulate a basic authenticationcredentials entry in the*  *// credentials provider.*

**return** credentials;   
 }   
 **returnsuper**.getCredentials(authscope);   
 }   
}

**privateclass** PreAuthAuthCache **extends** BasicAuthCache {  
 @Override   
 **public** AuthScheme get(HttpHost host){   
 **if**(hostNameMatches(host.getHostName())){   
 *// Simulate a cache entry for this host. This instructs*  *// HttpClient to use basic authentication for this host.*

**returnnew** BasicScheme();   
 }   
 **returnsuper**.get(host);   
 }   
}

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 45 |

This can be used as follows:

|  |
| --- |
| HttpClientBuilder builder = HttpClientBuilder.create(); |

|  |
| --- |
| ClientHttpRequestFactory requestFactory = |

|  |
| --- |
| **new** PreAuthHttpComponentsClientHttpRequestFactory(builder.build(), |

|  |
| --- |
| "api.some-host.com", **true**, "api", "my-key"); |

|  |
| --- |
| RestTemplate restTemplate =**new** RestTemplate(requestFactory); |

**Section 10.5: Using Basic Authentication with HttpComponent's HttpClient**

Using HttpClient as RestTemplate's underlying implementation to create HTTP requests allows for automatic

handling of basic authentication requests (an http 401 response) when interacting with APIs. This example shows

how to configure a RestTemplate to achieve this.

*// The credentials are stored here*

CredentialsProvider credsProvider =**new** BasicCredentialsProvider();

credsProvider.setCredentials(

*// AuthScope can be configured more extensively to restrict*

*// for which host/port/scheme/etc the credentials will be used.*

**new** AuthScope("somehost", AuthScope.ANY\_PORT),

|  |
| --- |
| **new** UsernamePasswordCredentials("username", "password")); |

|  |
| --- |
| *// Use the credentials provider* |

HttpClientBuilder builder = HttpClientBuilder.create();

|  |
| --- |
| builder.setDefaultCredentialsProvider(credsProvider); |

|  |
| --- |
| *// Configure the RestTemplate to use HttpComponent's HttpClient* |

ClientHttpRequestFactory requestFactory =

**new** HttpComponentsClientHttpRequestFactory(builder.build());

|  |  |
| --- | --- |
| RestTemplate restTemplate =**new** RestTemplate(requestFactory); |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 46 |

**Chapter 11: Task Execution and Scheduling**

**Section 11.1: Enable Scheduling**

Spring provides a useful task scheduling support. To enable it, just annotate any of your [@Configuration](http://docs.spring.io/spring-framework/docs/4.0.4.RELEASE/javadoc-api/org/springframework/context/annotation/Configuration.html) classes

with [@EnableScheduling](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableScheduling.html):

|  |
| --- |
| @Configuration |

|  |
| --- |
| @EnableScheduling |

|  |
| --- |
| **publicclass** MyConfig { |

|  |
| --- |
| *// Here it goes your configuration* |

|  |
| --- |
| } |

**Section 11.2: Cron expression**

A Cron expression consists of six sequential fields -

|  |
| --- |
| second, minute, hour, day of month, month, day(s) of week |

and is declared as follows

|  |
| --- |
| @Scheduled(cron ="\* \* \* \* \* \*") |

We can also set the [timezone](https://docs.oracle.com/cd/B13866_04/webconf.904/b10877/timezone.htm) as -

|  |
| --- |
| @Scheduled(cron="\* \* \* \* \* \*", zone="Europe/Istanbul") |

**Notes: -**

|  |
| --- |
| syntax means example explanation |

|  |
| --- |
| ------------------------------------------------------------------------------------ |

|  |
| --- |
| \* match any "\* \* \* \* \* \*" do always |

|  |
| --- |
| \*/x every x "\*/5 \* \* \* \* \*" do every five seconds |

|  |
| --- |
| ? no specification "0 0 0 25 12 ?" do every Christmas Day |

**Example: -**

|  |
| --- |
| syntax means |

|  |
| --- |
| ------------------------------------------------------------------------------------ |

|  |
| --- |
| "0 0 \* \* \* \*" the top of every hour of every day. |

|  |
| --- |
| "\*/10 \* \* \* \* \*" every ten seconds. |

|  |
| --- |
| "0 0 8-10 \* \* \*" 8, 9 and 10 o'clock of every day. |

|  |
| --- |
| "0 0/30 8-10 \* \* \*" 8:00, 8:30, 9:00, 9:30 and 10 o'clock every day. |

|  |
| --- |
| "0 0 9-17 \* \* MON-FRI" on the hour nine-to-five weekdays |

|  |
| --- |
| "0 0 0 25 12 ?" every Christmas Day at midnight |

A method declared with @Scheduled() is called explicitly for every matching case.

If we want some code to be executed when a cron expression is met, then we have to specify it in the annotation:

|  |
| --- |
| @Component |

|  |  |
| --- | --- |
| **publicclass** MyScheduler{ |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 47 |

|  |
| --- |
| @Scheduled(cron="\*/5 \* \* \* \* MON-FRI") |

|  |
| --- |
| **publicvoid** doSomething(){ |

|  |
| --- |
| *// this will execute on weekdays* |

|  |
| --- |
| } |

|  |
| --- |
| } |

If we want to print current time in our console for every after 5 seconds -

**import**org.slf4j.Logger;

**import**org.slf4j.LoggerFactory;

**import**org.springframework.scheduling.annotation.Scheduled;

|  |
| --- |
| **import**org.springframework.stereotype.Component; |

|  |
| --- |
| **import**java.text.SimpleDateFormat; |

|  |
| --- |
| **import**java.util.Date; |

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** Scheduler { |

|  |
| --- |
| **privatestaticfinal** Logger log = LoggerFactory.getLogger(Scheduler.**class**); |

|  |
| --- |
| **privatestaticfinal**SimpleDateFormat dateFormat =**new**SimpleDateFormat("HH:mm:ss"); |

|  |
| --- |
| @Scheduled(cron ="\*/5 \* \* \* \* \*") |

**publicvoid** currentTime(){

log.info("Current Time = {}", dateFormat.format(**new**Date()));

|  |
| --- |
| } |

|  |
| --- |
| } |

**Example using XML configuration:**

Example class:

|  |
| --- |
| **import**org.slf4j.Logger; |

|  |
| --- |
| **import**org.slf4j.LoggerFactory; |

|  |
| --- |
| **import**org.springframework.stereotype.Component; |

|  |
| --- |
| **import**java.text.SimpleDateFormat; |

|  |
| --- |
| **import**java.util.Date; |

|  |
| --- |
| @Component("schedulerBean") |

|  |
| --- |
| **publicclass** Scheduler { |

|  |
| --- |
| **privatestaticfinal** Logger log = LoggerFactory.getLogger(Scheduler.**class**); |

|  |
| --- |
| **privatestaticfinal**SimpleDateFormat dateFormat =**new**SimpleDateFormat("HH:mm:ss"); |

|  |
| --- |
| **publicvoid** currentTime(){ |

log.info("Current Time = {}", dateFormat.format(**new**Date()));

|  |
| --- |
| } |

|  |
| --- |
| } |

Example XML(task-context.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" |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 48 |

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

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

http://www.springframework.org/schema/beans/spring-beans-4.1.xsd

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

|  |
| --- |
| http://www.springframework.org/schema/task/spring-task-4.1.xsd"**>** |

|  |
| --- |
| **<task:scheduled-tasks**scheduler="scheduledTasks"**>** |

**<task:scheduled**ref="schedulerBean"method="currentTime"cron="\*/5 \* \* \* \* MON-FRI"**/>**

|  |
| --- |
| **</task:scheduled-tasks>** |

|  |
| --- |
| **<task:scheduler**id="scheduledTasks"**/>** |

|  |
| --- |
| **</beans>** |

**Section 11.3: Fixed delay**

If we want some code to be executed periodically after the execution which was before is finished, we should use

fixed delay (measured in milliseconds):

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** MyScheduler{ |

|  |
| --- |
| @Scheduled(fixedDelay=5000) |

|  |
| --- |
| **publicvoid** doSomething(){ |

|  |
| --- |
| *// this will execute periodically, after the one before finishes* |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Section 11.4: Fixed Rate**

If we want something to be executed periodically, this code will be triggered once per the value in milliseconds we

specify:

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** MyScheduler{ |

|  |
| --- |
| @Scheduled(fixedRate=5000) |

|  |
| --- |
| **publicvoid** doSomething(){ |

|  |
| --- |
| *// this will execute periodically* |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 49 |

**Chapter 12: Spring Lazy Initialization**

**Section 12.1: Example of Lazy Init in Spring**

The @Lazy allow us to instruct the IOC container to delay the initialization of a bean. By default, beans are

instantiated as soon as the IOC container is created, The @Lazy allow us to change this instantiation process.

lazy-init in spring is the attribute of bean tag. The values of lazy-init are true and false. If lazy-init is true, then that

bean will be initialized when a request is made to bean. This bean will not be initialized when the spring container is

initialized. If lazy-init is false then the bean will be initialized with the spring container initialization and this is the

default behavior.

**app-conf.xml**

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

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

|  |
| --- |
| xmlns:util="http://www.springframework.org/schema/util" |

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

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

|  |
| --- |
| http://www.springframework.org/schema/util |

|  |
| --- |
| http://www.springframework.org/schema/util/spring-util-3.0.xsd"**>** |

|  |
| --- |
| **<bean**id="testA"class="com.concretepage.A"**/>** |

|  |
| --- |
| **<bean**id="testB"class="com.concretepage.B"lazy-init="true"**/>** |

**A.java**

|  |
| --- |
| **package**com.concretepage; |

|  |
| --- |
| **publicclass** A { |

|  |
| --- |
| **public** A(){ |

|  |
| --- |
| System.out.println("Bean A is initialized"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

**B.java**

|  |
| --- |
| **package**com.concretepage; |

|  |
| --- |
| **publicclass** B { |

|  |
| --- |
| **public** B(){ |

|  |
| --- |
| System.out.println("Bean B is initialized"); |

|  |
| --- |
| } |

|  |
| --- |
| } |

**SpringTest.java**

|  |
| --- |
| **package**com.concretepage; |

|  |
| --- |
| **import**org.springframework.context.ApplicationContext; |

|  |
| --- |
| **import**org.springframework.context.support.ClassPathXmlApplicationContext; |

|  |
| --- |
| **publicclass** SpringTest { |

|  |
| --- |
| **publicstaticvoid** main(String[] args){ |

|  |
| --- |
| ApplicationContext context =**new** ClassPathXmlApplicationContext("app-conf.xml"); |

|  |
| --- |
| System.out.println("Feth bean B."); |

|  |
| --- |
| context.getBean("testB"); |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 50 |

**Output**

|  |
| --- |
| Bean A is initialized |

|  |
| --- |
| Feth bean B. |

|  |
| --- |
| Bean B is initialized |

**Section 12.2: For component scanning and auto-wiring**

@Component

@Lazy

**publicclass** Demo {

....

....

|  |
| --- |
| } |

|  |
| --- |
| @Component |

|  |
| --- |
| **publicclass** B { |

|  |
| --- |
| @Autowired |

@Lazy *// If this is not here, Demo will still get eagerly instantiated to satisfy this request.*

|  |
| --- |
| **private** Demo demo; |

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

}

**Section 12.3: Lazy initialization in the configuration class**

|  |
| --- |
| @Configuration |

|  |
| --- |
| *// @Lazy - For all Beans to load lazily* |

|  |
| --- |
| **publicclass** AppConf { |

|  |
| --- |
| @Bean |

|  |
| --- |
| @Lazy |

|  |
| --- |
| **public** Demo demo(){ |

|  |
| --- |
| **returnnew** Demo(); |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 51 |

**Chapter 13: Property Source**

**Section 13.1: Sample xml configuration using PropertyPlaceholderConfigurer**

|  |
| --- |
| **<bean**class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>** |

|  |
| --- |
| **<property**name="locations"**>** |

|  |
| --- |
| **<list>** |

|  |
| --- |
| **<value>**classpath:ReleaseBundle.properties**</value>** |

|  |
| --- |
| **</list>** |

|  |
| --- |
| **</bean>** |

**Section 13.2: Annotation**

Sample property file : nexus.properties

Sample property file content:

|  |
| --- |
| nexus.user=admin |

|  |
| --- |
| nexus.pass=admin |

|  |
| --- |
| nexus.rest.uri=http:*//xxx.xxx.xxx.xxx:xxxx/nexus/service/local/artifact/maven/content* |

Sample Context File xml configuration

|  |
| --- |
| **<context:property-placeholder**location="classpath:ReleaseBundle.properties"**/>** |

Sample Property Bean using annotations

|  |
| --- |
| @Component |

@PropertySource(value ={"classpath:nexus.properties"})

|  |
| --- |
| **publicclass** NexusBean { |

|  |
| --- |
| @Value("${"+ NexusConstants.NEXUS\_USER+"}") |

|  |
| --- |
| **private**String user; |

|  |
| --- |
| @Value("${"+ NexusConstants.NEXUS\_PASS+"}") |

|  |
| --- |
| **private**String pass; |

|  |
| --- |
| @Value("${"+ NexusConstants.NEXUS\_REST\_URI+"}") |

**private**String restUri;

|  |
| --- |
| } |

Sample Constant class

|  |
| --- |
| **publicclass** NexusConstants { |

|  |
| --- |
| **publicstaticfinal**String NexusConstants.NEXUS\_USER=""; |

|  |
| --- |
| **publicstaticfinal**String NexusConstants.NEXUS\_PASS=""; |

|  |
| --- |
| **publicstaticfinal**String NexusConstants.NEXUS\_REST\_URI=""; |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 52 |

**Chapter 14: Dependency Injection (DI) and Inversion of Control (IoC)**

**Section 14.1: Autowiring a dependency through Java configuration**

Constructor injection through Java configuration can also utilize autowiring, such as:

|  |
| --- |
| @Configuration |

|  |
| --- |
| **class** AppConfig { |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** Bar bar(){**returnnew** Bar();} |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** Foo foo(Bar bar){**returnnew** Foo(bar);} |

|  |
| --- |
| } |

**Section 14.2: Autowiring a dependency through XML configuration**

Dependencies can be autowired when using the component scan feature of the Spring framework. For autowiring

to work, the following XML configuration must be made:

|  |
| --- |
| **<context:annotation-config/>** |

|  |
| --- |
| **<context:component-scan**base-package="[base package]"**/>** |

where, base-**package** is the fully-qualified Java package within which Spring should perform component scan.

Constructor injection

Dependencies can be injected through the class constructor as follows:

|  |
| --- |
| @Component |

|  |
| --- |
| **class** Bar { ... } |

|  |
| --- |
| @Component |

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| **private** Bar bar; |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **public** Foo(Bar bar){**this**.bar= bar;} |

|  |
| --- |
| } |

Here, @Autowired is a Spring-specific annotation. Spring also supports [JSR-299](http://docs.oracle.com/javaee/6/tutorial/doc/giwhb.html) to enable application portability to

other Java-based dependency injection frameworks. This allows @Autowired to be replaced with @Inject as:

|  |
| --- |
| @Component |

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| **private** Bar bar; |

|  |
| --- |
| @Inject |

|  |
| --- |
| **public** Foo(Bar bar){**this**.bar= bar;} |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 53 |

Property injection

Dependencies can also be injected using setter methods as follows:

|  |
| --- |
| @Component |

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| **private** Bar bar; |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **publicvoid** setBar(Bar bar){**this**.bar= bar;} |

|  |
| --- |
| } |

Field injection

Autowiring also allows initializing fields within class instances directly, as follows:

|  |
| --- |
| @Component |

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **private** Bar bar; |

|  |
| --- |
| } |

For Spring versions 4.1+ you can use [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) for optional dependencies.

|  |
| --- |
| @Component |

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| @Autowired |

|  |
| --- |
| **private** Optional<Bar> bar; |

|  |
| --- |
| } |

The same approach can be used for constructor DI.

|  |
| --- |
| @Component |

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| **private** Optional<Bar> bar; |

|  |
| --- |
| @Autowired |

|  |
| --- |
| Foo(Optional<Bar> bar){ |

|  |
| --- |
| **this**.bar= bar; |

|  |
| --- |
| } |

|  |
| --- |
| } |

**Section 14.3: Injecting a dependency manually through XML configuration**

Consider the following Java classes:

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| **private** Bar bar; |

|  |
| --- |
| **publicvoid** foo(){ |

|  |
| --- |
| bar.baz(); |

|  |
| --- |
| } |

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 54 |

As can be seen, the class Foo needs to call the method baz on an instance of another class Bar for its method foo to

work successfully. Bar is said to be a dependency for Foo since Foo cannot work correctly without a Bar instance.

Constructor injection

When using XML configuration for Spring framework to define Spring-managed beans, a bean of type Foo can be

configured as follows:

|  |
| --- |
| **<bean**class="Foo"**>** |

|  |
| --- |
| **<constructor-arg>** |

|  |
| --- |
| **<bean**class="Bar"**/>** |

|  |
| --- |
| **</constructor-arg>** |

|  |
| --- |
| **</bean>** |

or, alternatively (more verbose):

|  |
| --- |
| **<bean**id="bar"class="bar"**/>** |

|  |
| --- |
| **<bean**class="Foo"**>** |

|  |
| --- |
| **<constructor-arg**ref="bar"**/>** |

|  |
| --- |
| **</bean>** |

In both cases, Spring framework first creates an instance of Bar and injects it into an instance of Foo. This example

assumes that the class Foo has a constructor that can take a Bar instance as a parameter, that is:

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| **private** Bar bar; |

|  |
| --- |
| **public** Foo(Bar bar){**this**.bar= bar;} |

|  |
| --- |
| } |

This style is known as **constructor injection** because the dependency (Bar instance) is being injected into through

the class constructor.

Property injection

Another option to inject the Bar dependency into Foo is:

|  |
| --- |
| **<bean**class="Foo"**>** |

|  |
| --- |
| **<property**name="bar"**>** |

|  |
| --- |
| **<bean**class="Bar"**/>** |

|  |
| --- |
| **</property>** |

|  |
| --- |
| **</bean>** |

or, alternatively (more verbose):

|  |
| --- |
| **<bean**id="bar"class="bar"**/>** |

|  |
| --- |
| **<bean**class="Foo"**>** |

|  |
| --- |
| **<property**name="bar"ref="bar"**/>** |

|  |
| --- |
| **</bean>** |

This requires the Foo class to have a setter method that accepts a Bar instance, such as:

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 55 |

|  |
| --- |
| **class** Foo { |

|  |
| --- |
| **private** Bar bar; |

|  |
| --- |
| **publicvoid** setBar(Bar bar){**this**.bar= bar;} |

|  |
| --- |
| } |

**Section 14.4: Injecting a dependency manually through Java configuration**

The same examples as shown above with XML configuration can be re-written with Java configuration as follows.

Constructor injection

|  |
| --- |
| @Configuration |

|  |
| --- |
| **class** AppConfig { |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** Bar bar(){**returnnew** Bar();} |

|  |
| --- |
| @Bean |

|  |
| --- |
| **public** Foo foo(){**returnnew** Foo(bar());} |

|  |
| --- |
| } |

Property injection

|  |
| --- |
| @Configuration |

**class** AppConfig {

@Bean

|  |
| --- |
| **public** Bar bar(){**returnnew** Bar();} |

|  |
| --- |
| @Bean |

**public** Foo foo(){

Foo foo =**new** Foo();

|  |
| --- |
| foo.setBar(bar()); |

|  |
| --- |
| **return** foo; |

}

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 56 |

**Chapter 15: JdbcTemplate**

The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration

over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them

to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the JdbcTemplate class are threadsafe once configured so it can be safely inject this shared reference

into multiple DAOs.

**Section 15.1: Basic Query methods**

Some of the queryFor\* methods available in JdbcTemplate are useful for simple sql statements that perform CRUD

operations.

**Querying for Date**

|  |
| --- |
| String sql ="SELECT create\_date FROM customer WHERE customer\_id = ?"; |

|  |
| --- |
| **int** storeId = jdbcTemplate.queryForObject(sql, java.util.Date.**class**, customerId); |

**Querying for Integer**

|  |
| --- |
| String sql ="SELECT store\_id FROM customer WHERE customer\_id = ?"; |

|  |
| --- |
| **int** storeId = jdbcTemplate.queryForObject(sql, Integer.**class**, customerId); |

OR

|  |
| --- |
| String sql ="SELECT store\_id FROM customer WHERE customer\_id = ?"; |

|  |
| --- |
| **int** storeId = jdbcTemplate.queryForInt(sql, customerId);*//Deprecated in spring-* |

|  |
| --- |
| *jdbc 4* |

**Querying for String**

|  |
| --- |
| String sql ="SELECT first\_Name FROM customer WHERE customer\_id = ?"; |

|  |
| --- |
| String firstName = jdbcTemplate.queryForObject(sql, String.**class**, customerId); |

**Querying for List**

|  |
| --- |
| String sql ="SELECT first\_Name FROM customer WHERE store\_id = ?"; |

|  |
| --- |
| List<String> firstNameList = jdbcTemplate.queryForList(sql, String.**class**, storeId); |

**Section 15.2: Query for List of Maps**

|  |
| --- |
| **int** storeId =1; |

|  |
| --- |
| DataSource dataSource = ... *//* |

|  |
| --- |
| JdbcTemplate jdbcTemplate =**new** JdbcTemplate(dataSource); |

|  |
| --- |
| String sql ="SELECT \* FROM customer WHERE store\_id = ?"; |

|  |
| --- |
| List<Map<String, Object>> mapList = jdbcTemplate.queryForList(sql, storeId); |

|  |
| --- |
| **for**(Map<String, Object> entryMap : mapList) |

|  |
| --- |
| { |

**for**(Entry<String, Object> entry : entryMap.entrySet())

{

System.out.println(entry.getKey()+" / "+ entry.getValue());

}

|  |  |
| --- | --- |
| System.out.println("---"); |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 57 |

|  |
| --- |
| } |

**Section 15.3: SQLRowSet**

|  |
| --- |
| DataSource dataSource = ... *//* |

JdbcTemplate jdbcTemplate =**new** JdbcTemplate(dataSource); String sql ="SELECT \* FROM customer";

|  |
| --- |
| SqlRowSet rowSet = jdbcTemplate.queryForRowSet(sql); |

|  |
| --- |
| **while**(rowSet.next()) |

{   
String firstName = rowSet.getString("first\_name"); String lastName = rowSet.getString("last\_name"); System.out.println("Vorname: "+ firstName);   
System.out.println("Nachname: "+ lastName);   
System.out.println("---”);

|  |
| --- |
| } |

OR

String sql ="SELECT \* FROM customer";

|  |
| --- |
| List<Customer> customerList = jdbcTemplate.query(sql, **new** RowMapper<Customer>(){ |

|  |
| --- |
| @Override |

**public** Customer mapRow(ResultSet rs, **int** rowNum)**throws**SQLException {  
 Customer customer =**new** Customer();  
 customer.setFirstName(rs.getString("first\_Name"));  
 customer.setLastName(rs.getString("first\_Name"));

|  |
| --- |
| customer.setEmail(rs.getString("email")); |

|  |
| --- |
| **return** customer; |

|  |
| --- |
| } |

|  |
| --- |
| }); |

**Section 15.4: Batch operations**

JdbcTemplate also provides convenient methods to execute batch operations.

**Batch Insert**

|  |
| --- |
| **final** ArrayList<Student> list =*// Get list of students to insert..* |

|  |
| --- |
| String sql ="insert into student (id, f\_name, l\_name, age, address) VALUES (?, ?, ?, ?, ?)" |

|  |
| --- |
| jdbcTemplate.batchUpdate(sql, **new** BatchPreparedStatementSetter(){ |

@Override   
 **publicvoid** setValues(PreparedStatement ps, **int** i)**throws**SQLException{ Student s = l.get(i);  
 ps.setString(1, s.getId());  
 ps.setString(2, s.getF\_name());  
 ps.setString(3, s.getL\_name());  
 ps.setInt(4, s.getAge());  
 ps.setString(5, s.getAddress());

|  |
| --- |
| } |

|  |
| --- |
| @Override |

**publicint** getBatchSize(){   
 **return** l.size();

|  |  |
| --- | --- |
| } |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 58 |

|  |
| --- |
| }); |

**Batch Update**

**final** ArrayList<Student> list =*// Get list of students to update..*

String sql ="update student set f\_name = ?, l\_name = ?, age = ?, address = ? where id = ?" jdbcTemplate.batchUpdate(sql, **new** BatchPreparedStatementSetter(){  
 @Override   
 **publicvoid** setValues(PreparedStatement ps, **int** i)**throws**SQLException{  
 Student s = l.get(i);  
 ps.setString(1, s.getF\_name());  
 ps.setString(2, s.getL\_name());  
 ps.setInt(3, s.getAge());  
 ps.setString(4, s.getAddress());  
 ps.setString(5, s.getId());

|  |
| --- |
| } |

|  |
| --- |
| @Override |

**publicint** getBatchSize(){   
 **return** l.size();   
 }   
});

There are further batchUpdate methods which accept List of object array as input parameters. These methods internally use BatchPreparedStatementSetter to set the values from the list of arrays into sql statement.

**Section 15.5: NamedParameterJdbcTemplate extension of JdbcTemplate**

The NamedParameterJdbcTemplate class adds support for programming JDBC statements using named parameters, as opposed to programming JDBC statements using only classic placeholder ( '?') arguments.

The NamedParameterJdbcTemplate class wraps a JdbcTemplate, and delegates to the wrapped JdbcTemplate to do much of its work.

|  |
| --- |
| DataSource dataSource = ... *//* |

|  |
| --- |
| NamedParameterJdbcTemplate jdbcTemplate =**new** NamedParameterJdbcTemplate(dataSource); |

|  |
| --- |
| String sql ="SELECT count(\*) FROM customer WHERE city\_name=:cityName"; |

|  |
| --- |
| Map<String, String> params =Collections.singletonMap("cityName", cityName); |

|  |  |
| --- | --- |
| **int** count = jdbcTemplate.queryForObject(sql, params, Integer.**class**); |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 59 |

**Chapter 16: SOAP WS Consumption**

**Section 16.1: Consuming a SOAP WS with Basic auth**

Create your own WSMessageSender:

**import**java.io.IOException;

|  |
| --- |
| **import**java.net.HttpURLConnection; |

|  |
| --- |
| **import**org.springframework.ws.transport.http.HttpUrlConnectionMessageSender; |

|  |
| --- |
| **import**sun.misc.BASE64Encoder; |

|  |
| --- |
| **publicclass** CustomWSMessageSender **extends** HttpUrlConnectionMessageSender{ |

|  |
| --- |
| @Override |

**protectedvoid** prepareConnection(HttpURLConnection connection)

|  |
| --- |
| **throws**IOException{ |

|  |
| --- |
| BASE64Encoder enc =**new** sun.misc.BASE64Encoder(); |

String userpassword ="yourUser:yourPassword";   
String encodedAuthorization = enc.encode( userpassword.getBytes());

|  |
| --- |
| connection.setRequestProperty("Authorization", "Basic "+ encodedAuthorization); |

|  |
| --- |
| **super**.prepareConnection(connection); |

}   
}

In your WS configuration class set the MessageSender you just created:

|  |  |  |
| --- | --- | --- |
| |  | | --- | | myWSClient.setMessageSender(**new** CustomWSMessageSender()); | |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 60 |

**Chapter 17: Spring profile**   
**Section 17.1: Spring Profiles allows to configure parts available for certain environment**   
Any @Component or @Configuration could be marked with @Profile annotation

|  |
| --- |
| @Configuration |

|  |
| --- |
| @Profile("production") |

|  |
| --- |
| **publicclass** ProductionConfiguration { |

|  |
| --- |
| *// ...* |

|  |
| --- |
| } |

The same in XML config

|  |
| --- |
| **<beans**profile="dev"**>** |

|  |
| --- |
| **<bean**id="dataSource"class="<some data source class>" /> |

|  |
| --- |
| **</beans>** |

Active profiles could be configured in the application.properties file

|  |
| --- |
| spring.profiles.active=dev,production |

or specified from command line

|  |
| --- |
| --spring.profiles.active=dev,hsqldb |

or in SpringBoot

|  |
| --- |
| SpringApplication.setAdditionalProfiles("dev"); |

It is possible to enable profiles in Tests using the annotation @ActiveProfiles("dev")

|  |  |
| --- | --- |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 61 |

**Chapter 18: Understanding the dispatcher-servlet.xml**

In Spring Web MVC, DispatcherServlet class works as the front controller. It is responsible for managing the flow of the spring MVC application.

DispatcherServlet is also like normal servlet need to be configured in web.xml

**Section 18.1: dispatcher-servlet.xml**

This is the important configuration file where we need to specify the ViewResolver and View components.

The context:component-scan element defines the base-package where DispatcherServlet will search the controller class.

Here, the InternalResourceViewResolver class is used for the ViewResolver.

The prefix+string returned by controller+suffix page will be invoked for the view component.

This xml file should be located inside the WEB-INF directory.

**<beans**xmlns="http://www.springframework.org/schema/beans" xmlns:context="http://www.springframework.org/schema/context" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"   
 xsi:schemaLocation="   
http://www.springframework.org/schema/beans   
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd http://www.springframework.org/schema/context

|  |
| --- |
| http://www.springframework.org/schema/context/spring-context-3.0.xsd"**>** |

|  |
| --- |
| **<context:component-scan**base-package="com.srinu.controller.Employee"**/>** |

|  |
| --- |
| **<bean** |

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

**Section 18.2: dispatcher servlet configuration in web.xml**

In this XML file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the HTML file will be forwarded to the DispatcherServlet.

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

|  |
| --- |
| **<web-app**version="2.5" |

|  |
| --- |
| xmlns="http://java.sun.com/xml/ns/javaee" |

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

|  |
| --- |
| xsi:schemaLocation="http://java.sun.com/xml/ns/javaee |

|  |
| --- |
| http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"**>** |

|  |
| --- |
| **<servlet>** |

|  |  |
| --- | --- |
| **<servlet-name>**spring**</servlet-name>** |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 62 |

|  |
| --- |
| **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>** |

|  |
| --- |
| **<load-on-startup>**1**</load-on-startup>** |

|  |
| --- |
| **</servlet>** |

|  |
| --- |
| **<servlet-mapping>** |

|  |
| --- |
| **<servlet-name>**spring**</servlet-name>** |

|  |
| --- |
| **<url-pattern>**\*.html**</url-pattern>** |

|  |
| --- |
| **</servlet-mapping>** |

|  |  |
| --- | --- |
| **</web-app>** |  |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | 63 |

**Credits**   
 Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to [web@petercv.com](mailto:web@petercv.com) for new content to be published or updated

|  |  |  |
| --- | --- | --- |
| [AdamIJK](https://stackoverflow.com/users/3913948/) | Chapter 12 |  |
| [Arlo](https://stackoverflow.com/users/197733/) | Chapter 4 |
| [bernie](https://stackoverflow.com/users/1030527/) | Chapter 10 |
| [Bond](https://stackoverflow.com/users/1910582/) | Chapter 7 |
| [CollinD](https://stackoverflow.com/users/5298696/) | Chapter 5 |
| [Constantine](https://stackoverflow.com/users/2267100/) | Chapters 5 and 6 |
| [DavidR](https://stackoverflow.com/users/1947251/) | Chapter 12 |
| [dimitrisli](https://stackoverflow.com/users/185723/) | Chapter 1 |
| [eltabo](https://stackoverflow.com/users/2165447/) | Chapter 10 |
| [Gautam Jose](https://stackoverflow.com/users/4921429/) | Chapter 13 |
| [guille11](https://stackoverflow.com/users/4091373/) | Chapters 11 and 16 |
| [Harshal Patil](https://stackoverflow.com/users/3244185/) | Chapter 5 |
| [Hitesh Kumar](https://stackoverflow.com/users/5821591/) | Chapter 1 |
| [ipsi](https://stackoverflow.com/users/415877/) | Chapter 1 |
| [JamesENL](https://stackoverflow.com/users/2357233/) | Chapter 5 |
| [Johir](https://stackoverflow.com/users/4251301/) | Chapter 11 |
| [manish](https://stackoverflow.com/users/1126526/) | Chapter 14 |
| [Moshe Arad](https://stackoverflow.com/users/6793312/) | Chapter 12 |
| [mszymborski](https://stackoverflow.com/users/5339966/) | Chapter 5 |
| [nicholas.hauschild](https://stackoverflow.com/users/343568/) | Chapter 9 |
| [Panther](https://stackoverflow.com/users/3617127/) | Chapters 1, 6 and 13 |
| [Praneeth Ramesh](https://stackoverflow.com/users/2197837/) | Chapter 8 |
| [Rajanikanta Pradhan](https://stackoverflow.com/users/6380454/) | Chapters 1 and 2 |
| [Sergii Bishyr](https://stackoverflow.com/users/5604676/) | Chapter 14 |
| [Setu](https://stackoverflow.com/users/3365479/) | Chapter 15 |
| [smichel](https://stackoverflow.com/users/7089830/) | Chapter 15 |
| [Srinivas Gadilli](https://stackoverflow.com/users/5815832/) | Chapter 18 |
| [StanislavL](https://stackoverflow.com/users/301607/) | Chapters 5, 7, 8, 10, 15 and 17 |
| [Stefan Isele](https://stackoverflow.com/users/3608192/) | Chapter 5 |
| [Taylor](https://stackoverflow.com/users/229743/) | Chapter 6 |
| [Tim Tong](https://stackoverflow.com/users/5818270/) | Chapters 5 and 6 |
| [walsh](https://stackoverflow.com/users/4101415/) | Chapter 3 |
| [xpadro](https://stackoverflow.com/users/1900288/) | Chapter 6 |
| [Xtreme Biker](https://stackoverflow.com/users/1199132/) | Chapter 11 |
| [GoalKicker.com – Spring® Framework Notes for Professionals](https://goalkicker.com/) | | 64 |

**You may also like**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |