# General Questions:

## What is DI?

It is a process whereby objects define their

dependencies, that is, the other objects they work with, only through constructor arguments, arguments

to a factory method, or properties that are set on the object instance after it is constructed or returned

from a factory method. The container then *injects* those dependencies when it creates the bean. This

process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself

controlling the instantiation or location of its dependencies by using direct construction of classes, or a

mechanism such as the *Service Locator* pattern.

## IOC in Spring? beanFActory vs ApplicationFActory?

The org.springframework.beans and org.springframework.context packages are the

basis for Spring Framework’s IoC container. The BeanFactory interface provides an advanced

configuration mechanism capable of managing any type of object. ApplicationContext is a subinterface

of BeanFactory. It adds easier integration with Spring’s AOP features; message resource

handling (for use in internationalization), event publication; and application-layer specific contexts such

as the WebApplicationContext for use in web applications.

In short, the BeanFactory provides the configuration framework and basic functionality, and the

ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is

a complete superset of the BeanFactory,

## Spring IOC container?

The interface org.springframework.context.ApplicationContext represents the Spring IoC

container and is responsible for instantiating, configuring, and assembling the aforementioned beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading

configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java

code. It allows you to express the objects that compose your application and the rich interdependencies

between such objects.

The following diagram is a high-level view of how Spring works. Your application classes are combined

with configuration metadata so that after the ApplicationContext is created and initialized, you have

a fully configured and executable system or application.

*Figure 3.1. The Spring IoC container*



## Ways to define metadata?

XML based

<?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">**

**<bean id**=**"..." class**=**"...">**

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

**</bean>**

**<bean id**=**"..." class**=**"...">**

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

**</bean>**

*<!-- more bean definitions go here -->*

**</beans>**

The id attribute is a string that you use to identify the individual bean definition. The class attribute

defines the type of the bean and uses the fully qualified classname.

Annotation based

Java based

## How to instantiate a Spring IOC container?

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {***"services.xml"***, ***"daos.xml"***});

The ApplicationContext is the interface for an advanced factory capable of maintaining a registry

of different beans and their dependencies. Using the method T getBean(String name, Class<T>

requiredType) you can retrieve instances of your beans.

The ApplicationContext enables you to read bean definitions and access them as follows:

*// create and configure beans*

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {***"services.xml"***, ***"daos.xml"***});

*// retrieve configured instance*

PetStoreService service = context.getBean(***"petStore"***, PetStoreService.**class**);

*// use configured instance*

List<String> userList = service.getUsernameList();

## How to register beans that are not created by Spring?

ApplicationContext implementations also permit the registration of existing objects that are

created outside the container, by users. This is done by accessing the ApplicationContext’s

BeanFactory via the method getBeanFactory() which returns the BeanFactory implementation

DefaultListableBeanFactory. DefaultListableBeanFactory supports this registration

through the methods registerSingleton(..) and registerBeanDefinition(..). However,

typical applications work solely with beans defined through metadata bean definitions.

## What is BEanDefiniton?

A Spring IoC container manages one or more *beans*. These beans are created with the configuration

metadata that you supply to the container, for example, in the form of XML <bean/> definitions.

Within the container itself, these bean definitions are represented as BeanDefinition objects, which

contain (among other information) the following metadata:

*A package-qualified class name:* typically the actual implementation class of the bean being defined.

• Bean behavioral configuration elements, which state how the bean should behave in the container

(scope, lifecycle callbacks, and so forth).

• References to other beans that are needed for the bean to do its work; these references are also

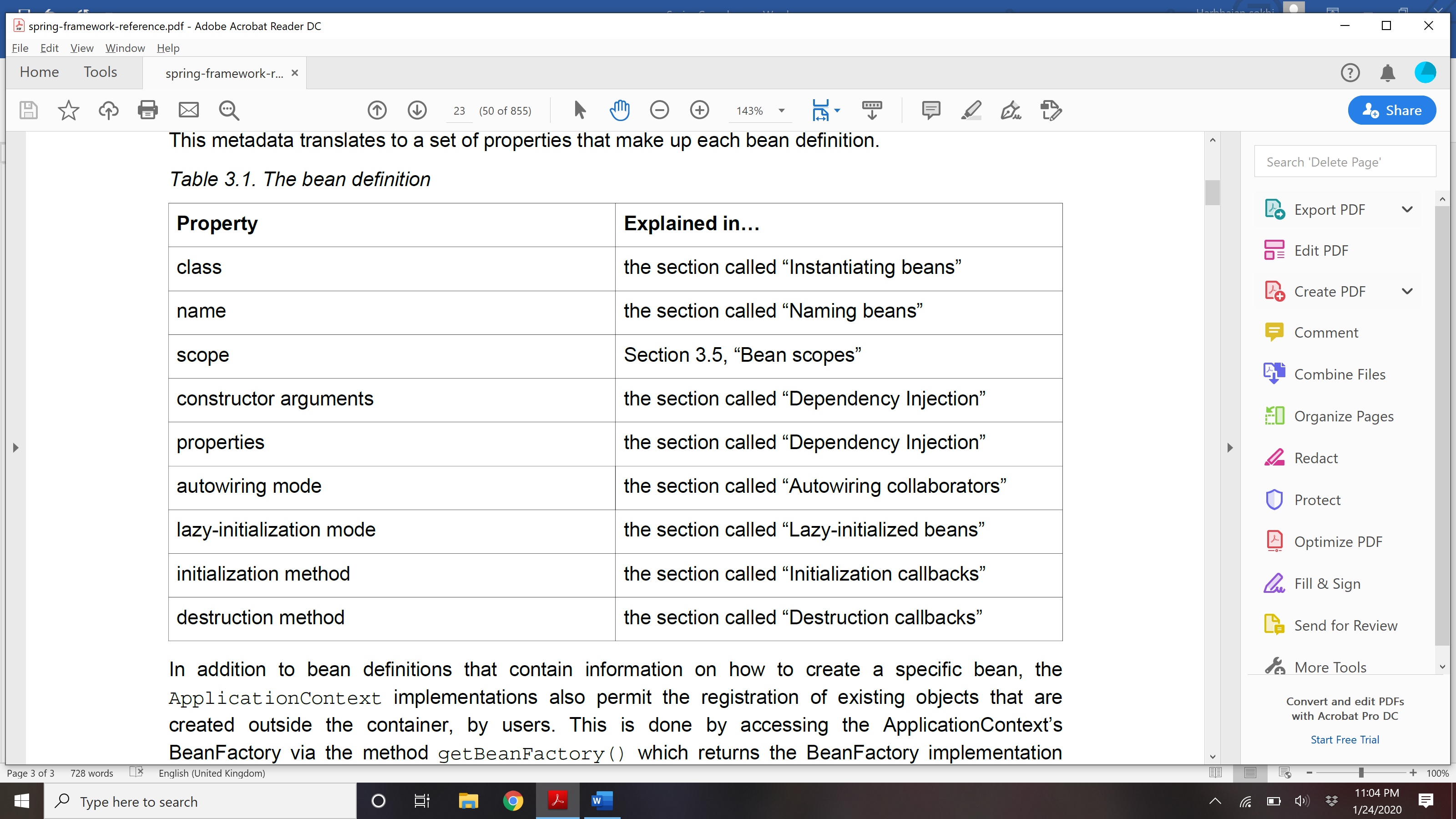
called *collaborators* or *dependencies*.

• Other configuration settings to set in the newly created object, for example, the number of connections

to use in a bean that manages a connection pool, or the size limit of the pool.

This metadata translates to a set of properties that make up each bean definition.

*Table 3.1. The bean definition*



## How spring instantiate beans?

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be

instantiated in the class attribute of the <bean/> element. This class attribute, which internally is a

Class property on a BeanDefinition instance, is usually mandatory. (For exceptions, see the section

called “Instantiation using an instance factory method” and Section 3.7, “Bean definition inheritance”.)

You use the Class property in one of two ways:

• Typically, to specify the bean class to be constructed in the case where the container itself directly

creates the bean by calling its constructor reflectively, somewhat equivalent to Java code using the

new operator.

• To specify the actual class containing the static factory method that will be invoked to create the

object, in the less common case where the container invokes a static *factory* method on a class

to create the bean. The object type returned from the invocation of the static factory method may

be the same class or another class entirely.

**Instantiation with a constructor**

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

**<bean name**=**"anotherExample" class**=**"examples.ExampleBeanTwo"/>**

**Instantiation with a static factory method**

**<bean id**=**"clientService"**

**class**=**"examples.ClientService"**

**factory-method**=**"createInstance"/>**

**public class** ClientService {

**private static** ClientService clientService = **new** ClientService();

**private** ClientService() {}

**public static** ClientService createInstance() {

**return** clientService;

}

}

**Instantiation using an instance factory method**

To use this

mechanism, leave the class attribute empty, and in the factory-bean attribute, specify the name of abean in the current (or parent/ancestor) container that contains the instance method that is to be invoked

to create the object. Set the name of the factory method itself with the factory-method attribute.

*<!-- the factory bean, which contains a method called createInstance() -->*

**<bean id**=**"serviceLocator" class**=**"examples.DefaultServiceLocator">**

*<!-- inject any dependencies required by this locator bean -->*

**</bean>**

*<!-- the bean to be created via the factory bean -->*

**<bean id**=**"clientService"**

**factory-bean**=**"serviceLocator"**

**factory-method**=**"createClientServiceInstance"/>**

**public class** DefaultServiceLocator {

**private static** ClientService clientService = **new** ClientServiceImpl();

**private** DefaultServiceLocator() {}

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

}

## Type of dependency Injection- constructor and setter?

## Constructor DI vs Setter DI?

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use

constructors for *mandatory dependencies* and setter methods or configuration methods for

*optional dependencies*. Note that use of the @Required annotation on a setter method can be

used to make the property a required dependency.

The Spring team generally advocates constructor injection as it enables one to implement

application components as *immutable objects* and to ensure that required dependencies are not

null. Furthermore constructor-injected components are always returned to client (calling) code

in a fully initialized state. As a side note, a large number of constructor arguments is a *bad code*

*smell*, implying that the class likely has too many responsibilities and should be refactored to better

address proper separation of concerns.

Setter injection should primarily only be used for optional dependencies that can be assigned

reasonable default values within the class. Otherwise, not-null checks must be performed

everywhere the code uses the dependency. One benefit of setter injection is that setter methods

make objects of that class amenable to reconfiguration or re-injection later. Management through

JMX MBeans is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with

third-party classes for which you do not have the source, the choice is made for you. For example,

if a third-party class does not expose any setter methods, then constructor injection may be the

only available form of DI.

## Lazy Initiatialization?

By default, ApplicationContext implementations eagerly create and configure all singleton beans

as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the

configuration or surrounding environment are discovered immediately, as opposed to hours or even

days later. When this behavior is *not* desirable, you can prevent pre-instantiation of a singleton bean by

marking the bean definition as lazy-initialized. A lazy-initialized bean tells the IoC container to create a

bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element; for example:

Spring Framework Reference Documentation

5.0.0.M1 Spring Framework 43

**<bean id**=**"lazy" class**=**"com.foo.ExpensiveToCreateBean" lazy-init**=**"true"/>**

**<bean name**=**"not.lazy" class**=**"com.foo.AnotherBean"/>**

When the preceding configuration is consumed by an ApplicationContext, the bean named lazy

is not eagerly pre-instantiated when the ApplicationContext is starting up, whereas the not.lazy

bean is eagerly pre-instantiated.

**However, when a lazy-initialized bean is a dependency of a singleton bean that is *not* lazy-initialized,**

**the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the**

**singleton’s dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is**

**not lazy-initialized.**

You can also control lazy-initialization at the container level by using the default-lazy-init attribute

on the <beans/> element; for example:

**<beans default-lazy-init**=**"true">**

*<!-- no beans will be pre-instantiated... -->*

**</beans>**

## Explain refBean, collection, property etc?

**Constructor :**

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

**<constructor-arg type**=**"int" value**=**"7500000"/>**

**<constructor-arg type**=**"java.lang.String" value**=**"42"/>**

**</bean>**

**For same type:**

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

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

Giving name for ambiguity:

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

**<constructor-arg name**=**"years" value**=**"7500000"/>**

**<constructor-arg name**=**"ultimateAnswer" value**=**"42"/>**

**</bean>**

**Setter DI:**

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

*<!-- setter injection using the nested ref element -->*

**<property name**=**"beanOne">**

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

**</property>**

*<!-- setter injection using the neater ref attribute -->*

**<property name**=**"beanTwo" ref**=**"yetAnotherBean"/>**

**<property name**=**"integerProperty" value**=**"1"/>**

**</bean>**

**<bean id**=**"anotherExampleBean" class**=**"examples.AnotherBean"/>**

**<bean id**=**"yetAnotherBean" class**=**"examples.YetAnotherBean"/>**

Constructor with ref bean:

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

*<!-- constructor injection using the nested ref element -->*

**<constructor-arg>**

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

**</constructor-arg>**

*<!-- constructor injection using the neater ref attribute -->*

**<constructor-arg ref**=**"yetAnotherBean"/>**

**<constructor-arg type**=**"int" value**=**"1"/>**

**</bean>**

**<bean id**=**"anotherExampleBean" class**=**"examples.AnotherBean"/>**

**<bean id**=**"yetAnotherBean" class**=**"examples.YetAnotherBean"/>**

**Util properties:**

**<bean id**=**"mappings"**

**class**=**"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">**

*<!-- typed as a java.util.Properties -->*

**<property name**=**"properties">**

**<value>**

jdbc.driver.className=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mydb

**</value>**

**</property>**

**</bean>**

**Also read about idref, innerbean, collection, parent bean, nul and empty string, depende-on**

## Autowiring ?

The Spring container can *autowire* relationships between collaborating beans. You can allow Spring

to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the

ApplicationContext. Autowiring has the following advantages:

• Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other

mechanisms such as a bean template discussed elsewhere in this chapter are also valuable in this

regard.)

• Autowiring can update a configuration as your objects evolve. For example, if you need to add a

dependency to a class, that dependency can be satisfied automatically without you needing to modify

the configuration. Thus autowiring can be especially useful during development, without negating the

option of switching to explicit wiring when the code base becomes more stable.

When using XML-based configuration metadata 10, you specify autowire mode for a bean definition

with the autowire attribute of the <bean/> element. The autowiring functionality has four modes. You

specify autowiring *per* bean and thus can choose which ones to autowire.

*Table 3.2. Autowiring modes*

**Mode Explanation**

**no** (Default) No autowiring. Bean references must

be defined via a ref element. Changing the

default setting is not recommended for larger

deployments, because specifying collaborators

explicitly gives greater control and clarity. To

some extent, it documents the structure of a

system.

**byName** Autowiring by property name. Spring looks for

a bean with the same name as the property

that needs to be autowired. For example, if a

bean definition is set to autowire by name, and

it contains a *master* property (that is, it has a

*setMaster(..)* method), Spring looks for a bean

definition named master, and uses it to set the

property.

**byType** Allows a property to be autowired if exactly one

bean of the property type exists in the container.

If more than one exists, a fatal exception is

thrown, which indicates that you may not use

*byType* autowiring for that bean. If there are no

matching beans, nothing happens; the property

is not set.

**constructor** Analogous to *byType*, but applies to constructor

arguments. If there is not exactly one bean of

the constructor argument type in the container, a

fatal error is raised.

## 12) disadvantages and limitation of autowiring

1) Explicit dependencies in property and constructor-arg settings always override autowiring..

2) Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful

to avoid guessing in case of ambiguity that might have unexpected results, the relationships between

your Spring-managed objects are no longer documented explicitly.

3) Multiple bean definitions within the container may match the type specified by the setter method

or constructor argument to be autowired.

## How to exclude autowiring:

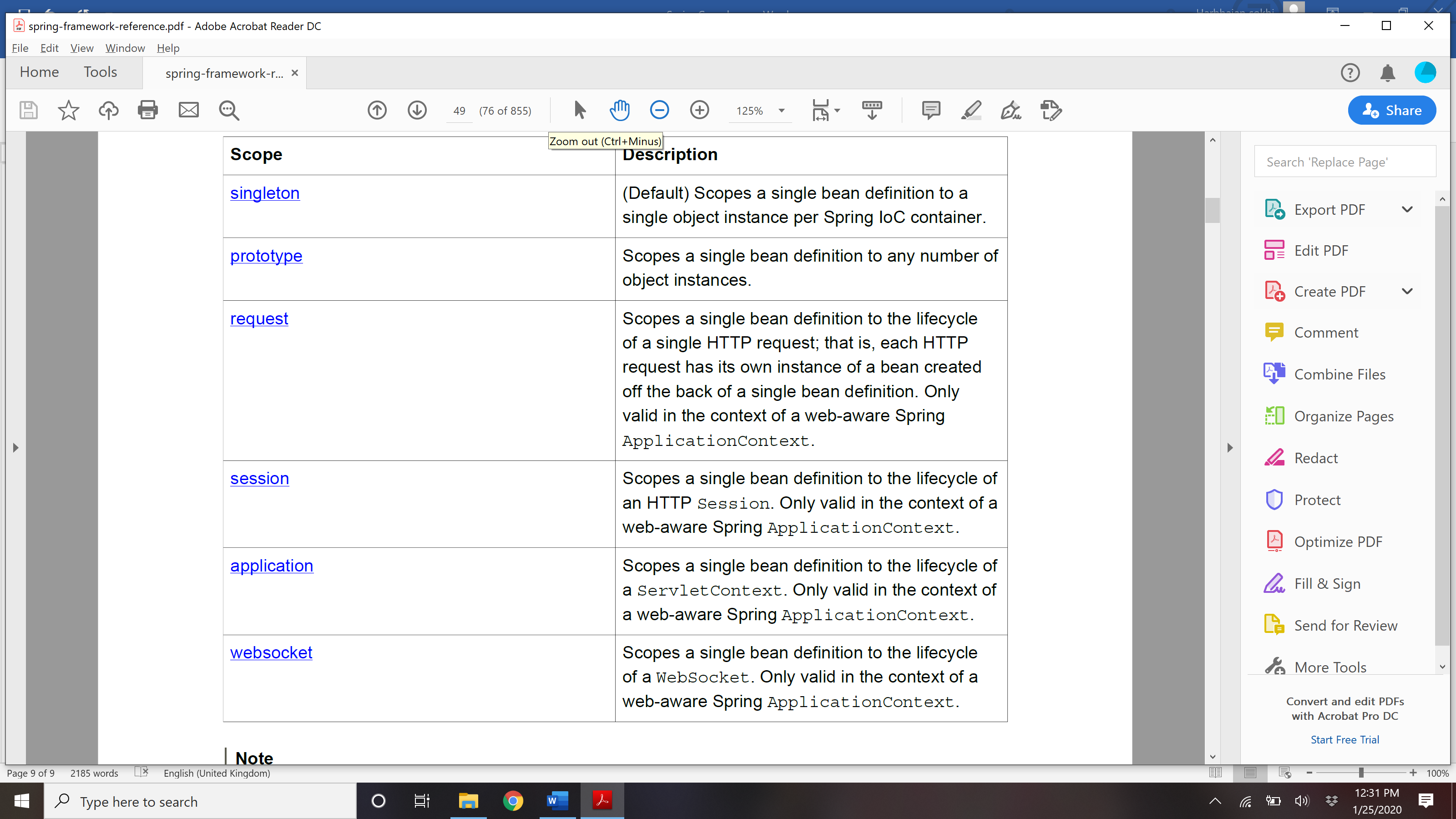
autowire-candidate = false on bean

default-autowire-candidates= fasle on top of bean declaration

## Look up Method or method Injection?

## Arbitrary Method replacement?

## What are the bean scopes?



Spring’s concept of a singleton bean differs from the Singleton pattern as defined in the Gang of Four

(GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one *and only*

*one* instance of a particular class is created *per ClassLoader*. The scope of the Spring singleton is best

described as *per container and per bean*. This means that if you define one bean for a particular class

in a single Spring container, then the Spring container creates one *and only one* instance of the class

defined by that bean definition.

Prototype:

The following example defines a bean as a prototype in XML:

**<bean id**=**"accountService" class**=**"com.foo.DefaultAccountService" scope**=**"prototype"/>**

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the

container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client,

with no further record of that prototype instance. Thus, although *initialization* lifecycle callback methods

are called on all objects regardless of scope, in the case of prototypes, configured *destruction* lifecycle

callbacks are *not* called. The client code must clean up prototype-scoped objects and release expensive

resources that the prototype bean(s) are holding. To get the Spring container to release resources held

by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans

that need to be cleaned up.

In some respects, the Spring container’s role in regard to a prototype-scoped bean is a replacement

for the Java new operator. All lifecycle management past that point must be handled by the client. (For

details on the lifecycle of a bean in the Spring container, see the section called “Lifecycle callbacks”.)

**Singleton beans with prototype-bean dependencies**

When you use singleton-scoped beans with dependencies on prototype beans, be aware that

*dependencies are resolved at instantiation time*. Thus if you dependency-inject a prototype-scoped bean

into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into

the singleton bean. The prototype instance is the sole instance that is ever supplied to the singletonscoped

bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototypescoped

bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your

singleton bean, because that injection occurs only *once*, when the Spring container is instantiating the

singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype

bean at runtime more than once, see the section called “Method injection”.

Other scopes:

The request, session, application, and websocket scopes are *only* available

if you use a web-aware Spring ApplicationContext implementation (such as

XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers

Spring Framework Reference Documentation

5.0.0.M1 Spring Framework 52

such as the ClassPathXmlApplicationContext, an IllegalStateException will be thrown

complaining about an unknown bean scope.

## How to create a custom scope?

## Bean Lifecycle?

When container starts – a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some post-initialization steps to get it into a usable state. Same bean life cycle is for [*spring boot*](https://howtodoinjava.com/spring-boot-tutorials/) applications as well.

After that, when the bean is no longer required, it will be removed from the IoC container.

Spring bean factory is responsible for managing the life cycle of beans created through spring container.

#### 1.1. Life cycle callbacks

Spring bean factory controls the creation and destruction of beans. To execute some custom code, it provides the call back methods which can be categorized broadly in two groups:

* **Post-initialization** call back methods
* **Pre-destruction** call back methods

#### 1.1. Life cycle in diagram

Spring Bean Life Cycle

## 2. Life cycle callback methods

Spring framework provides following **4 ways for controlling life cycle events** of a bean:

1. InitializingBean and DisposableBean callback interfaces
2. \*Aware interfaces for specific behavior
3. Custom init() and destroy() methods in bean configuration file
4. @PostConstruct and @PreDestroy annotations

#### 2.1. InitializingBean and DisposableBean

The [org.springframework.beans.factory.InitializingBean](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/beans/factory/InitializingBean.html" \o "InitializingBean) interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container.

The InitializingBean interface specifies a single method:

|  |
| --- |
| InitializingBean.java |
| void afterPropertiesSet() throws Exception; |

This is not a preferrable way to initialize the bean because it tightly couple your bean class with spring container. A better approach is to use “init-method” attribute in bean definition in applicationContext.xml file.

Similarly, implementing the [org.springframework.beans.factory.DisposableBean](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/beans/factory/DisposableBean.html" \o "DisposableBean) interface allows a bean to get a callback when the container containing it is destroyed.

The DisposableBean interface specifies a single method:

|  |
| --- |
| DisposableBean.java |
| void destroy() throws Exception;    A sample bean implementing above interfaces would look like this:      package com.howtodoinjava.task;    import org.springframework.beans.factory.DisposableBean;  import org.springframework.beans.factory.InitializingBean;    public class DemoBean implements InitializingBean, DisposableBean  {      //Other bean attributes and methods        @Override      public void afterPropertiesSet() throws Exception      {          //Bean initialization code      }        @Override      public void destroy() throws Exception      {          //Bean destruction code      }  } |

#### 2.2. \*Aware interfaces for specific behavior

Spring offers a range of \*Aware interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. Each interface will require you to implement a method to inject the dependency in bean.

These interfaces can be summarized as :

|  |  |  |
| --- | --- | --- |
| **AWARE INTERFACE** | **METHOD TO OVERRIDE** | **PURPOSE** |
| ApplicationContextAware | void setApplicationContext (ApplicationContext applicationContext) throws BeansException; | Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in. |
| ApplicationEventPublisherAware | void setApplicationEventPublisher (ApplicationEventPublisher applicationEventPublisher); | Set the ApplicationEventPublisher that this object runs in. |
| BeanClassLoaderAware | void setBeanClassLoader (ClassLoader classLoader); | Callback that supplies the bean class loader to a bean instance. |
| BeanFactoryAware | void setBeanFactory (BeanFactory beanFactory) throws BeansException; | Callback that supplies the owning factory to a bean instance. |
| BeanNameAware | void setBeanName(String name); | Set the name of the bean in the bean factory that created this bean. |
| BootstrapContextAware | void setBootstrapContext (BootstrapContext bootstrapContext); | Set the BootstrapContext that this object runs in. |
| LoadTimeWeaverAware | void setLoadTimeWeaver (LoadTimeWeaver loadTimeWeaver); | Set the LoadTimeWeaver of this object’s containing ApplicationContext. |
| MessageSourceAware | void setMessageSource (MessageSource messageSource); | Set the MessageSource that this object runs in. |
| NotificationPublisherAware | void setNotificationPublisher (NotificationPublisher notificationPublisher); | Set the NotificationPublisher instance for the current managed resource instance. |
| PortletConfigAware | void setPortletConfig (PortletConfig portletConfig); | Set the PortletConfig this object runs in. |
| PortletContextAware | void setPortletContext (PortletContext portletContext); | Set the PortletContext that this object runs in. |
| ResourceLoaderAware | void setResourceLoader (ResourceLoader resourceLoader); | Set the ResourceLoader that this object runs in. |
| ServletConfigAware | void setServletConfig (ServletConfig servletConfig); | Set the ServletConfig that this object runs in. |
| ServletContextAware | void setServletContext (ServletContext servletContext); | Set the ServletContext that this object runs in. |

Java program to show usage of aware interfaces to control string bean life cycle.

|  |
| --- |
| DemoBean.java |
| package com.howtodoinjava.task;    import org.springframework.beans.BeansException;  import org.springframework.beans.factory.BeanClassLoaderAware;  import org.springframework.beans.factory.BeanFactory;  import org.springframework.beans.factory.BeanFactoryAware;  import org.springframework.beans.factory.BeanNameAware;  import org.springframework.context.ApplicationContext;  import org.springframework.context.ApplicationContextAware;  import org.springframework.context.ApplicationEventPublisher;  import org.springframework.context.ApplicationEventPublisherAware;  import org.springframework.context.MessageSource;  import org.springframework.context.MessageSourceAware;  import org.springframework.context.ResourceLoaderAware;  import org.springframework.context.weaving.LoadTimeWeaverAware;  import org.springframework.core.io.ResourceLoader;  import org.springframework.instrument.classloading.LoadTimeWeaver;  import org.springframework.jmx.export.notification.NotificationPublisher;  import org.springframework.jmx.export.notification.NotificationPublisherAware;    public class DemoBean implements ApplicationContextAware,          ApplicationEventPublisherAware, BeanClassLoaderAware, BeanFactoryAware,          BeanNameAware, LoadTimeWeaverAware, MessageSourceAware,          NotificationPublisherAware, ResourceLoaderAware  {      @Override      public void setResourceLoader(ResourceLoader arg0) {          // TODO Auto-generated method stub      }        @Override      public void setNotificationPublisher(NotificationPublisher arg0) {          // TODO Auto-generated method stub        }        @Override      public void setMessageSource(MessageSource arg0) {          // TODO Auto-generated method stub      }        @Override      public void setLoadTimeWeaver(LoadTimeWeaver arg0) {          // TODO Auto-generated method stub      }        @Override      public void setBeanName(String arg0) {          // TODO Auto-generated method stub      }        @Override      public void setBeanFactory(BeanFactory arg0) throws BeansException {          // TODO Auto-generated method stub      }        @Override      public void setBeanClassLoader(ClassLoader arg0) {          // TODO Auto-generated method stub      }        @Override      public void setApplicationEventPublisher(ApplicationEventPublisher arg0) {          // TODO Auto-generated method stub      }        @Override      public void setApplicationContext(ApplicationContext arg0)              throws BeansException {          // TODO Auto-generated method stub      }  } |

#### 2.3. Custom init() and destroy() methods

The default init and destroy methods in bean configuration file can be defined in two ways:

* **Bean local definition** applicable to a single bean
* **Global definition** applicable to all beans defined in beans context

##### 2.3.1. Bean local definition

Local definition is given as below.

|  |
| --- |
| beans.xml |
| <beans>        <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"                      init-method="customInit"                      destroy-method="customDestroy"></bean>    </beans> |

##### 2.3.2. Global definition

Where as global definition is given as below. These methods will be invoked for all bean definitions given under <beans> tag. They are useful when you have a pattern of defining common method names such as init() and destroy() for all your beans consistently. This feature helps you in not mentioning the init and destroy method names for all beans independently.

|  |
| --- |
| <beans default-init-method="customInit" default-destroy-method="customDestroy">            <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"></bean>    </beans> |

Java program to show methods configured in bean XML configuration file.

|  |
| --- |
| DemoBean.java |
| package com.howtodoinjava.task;    public class DemoBean  {      public void customInit()      {          System.out.println("Method customInit() invoked...");      }        public void customDestroy()      {          System.out.println("Method customDestroy() invoked...");      }  } |

#### 2.4. @PostConstruct and @PreDestroy

Spring 2.5 onwards, you can use annotations also for specifying life cycle methods using @PostConstruct and @PreDestroy annotations.

* @PostConstruct annotated method will be invoked after the bean has been constructed using default constructor and just before it’s instance is returned to requesting object.
* @PreDestroy annotated method is called just before the bean is about be destroyed inside bean container.

Java program to show usage of **annotation configuration** to control using annotations.

|  |
| --- |
| package com.howtodoinjava.task;    import javax.annotation.PostConstruct;  import javax.annotation.PreDestroy;    public class DemoBean  {      @PostConstruct      public void customInit()      {          System.out.println("Method customInit() invoked...");      }        @PreDestroy      public void customDestroy()      {          System.out.println("Method customDestroy() invoked...");      }  } |

## Sequence of lifecycle methods?

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are

called as follows:

• Methods annotated with @PostConstruct

• afterPropertiesSet() as defined by the InitializingBean callback interface

• A custom configured init() method

Destroy methods are called in the same order:

• Methods annotated with @PreDestroy

• destroy() as defined by the DisposableBean callback interface

• A custom configured destroy() method

## Example of beanPostProcessor?

@Autowired, @Inject, @Resource, and @Value annotations are handled by Spring

BeanPostProcessor implementations which in turn means that you *cannot* apply these

annotations within your own BeanPostProcessor or BeanFactoryPostProcessor types (if

any). These types must be 'wired up' explicitly via XML or using a Spring @Bean method.

## BeanPostProcessor?

## @ Primary

Because autowiring by type may lead to multiple candidates, it is often necessary to have more

control over the selection process. One way to accomplish this is with Spring’s @Primary annotation.

@Primary indicates that a particular bean should be given preference when multiple beans are

candidates to be autowired to a single-valued dependency. If exactly one 'primary' bean exists among

the candidates, it will be the autowired value.

@Configuration

**public class** MovieConfiguration {

@Bean

**@Primary**

**public** MovieCatalog firstMovieCatalog() { ... }

@Bean

**public** MovieCatalog secondMovieCatalog() { ... }

*// ...*

}

With such configuration, the following MovieRecommender will be autowired with the

firstMovieCatalog.

## @Qualifier vs @ Primary

Read [@Primary](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Primary.html) as the "default".

If a bean has @Autowired *without* any @Qualifier, and multiple beans of the type exist, the candidate bean marked @Primary will be chosen, i.e. it is the default selection when no other information is available, i.e. when @Qualifier is missing.

A good use case is that initially you only had one bean of the type, so none of the code used @Qualifier. When you then add another bean, you then also add @Qualifier to both the old and the new bean, so any @Autowired can choose which one it wants. By also adding @Primary to the old original bean, you don't have to add @Qualifier to all the existing @Autowired. They are "grandfathered" in, so to speak.

## @Qualifier

In Spring, @Qualifier means, which bean is qualify to autowired on a field. See following scenario :

## **Autowiring Example**

See below example, it will autowired a “person” bean into customer’s person property.

package com.mkyong.common;

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

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

public class Customer {

@Autowired

private Person person;

//...

}

Copy

But, two similar beans “com.mkyong.common.Person” are declared in bean configuration file. Will Spring know which person bean should autowired?

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

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

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

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

<bean

class ="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

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

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

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

</bean>

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

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

</bean>

</beans>

Copy

When you run above example, it hits below exception :

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

No unique bean of type [com.mkyong.common.Person] is defined:

expected single matching bean but found 2: [personA, personB]

Copy

## **@Qualifier Example**

To fix above problem, you need **@Quanlifier** to tell Spring about which bean should autowired.

package com.mkyong.common;

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

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

public class Customer {

@Autowired

@Qualifier("personA")

private Person person;

//...

}

Copy

In this case, bean “personA” is autowired.

Customer [person=Person [name=mkyongA]]

## @Resource vs @Inject vs @Autowire

Resource and inject are JSR specification.

Autowire is spring specification

The behaviour of *@Autowired* annotation is similar to the *@Inject* annotation. The only difference is that the *@Autowired* annotation is part of the Spring framework. This annotation has the same execution paths as the *@Inject* annotation, listed in order of precedence:

1. Match by Type
2. Match by Qualifier
3. Match by Name

These execution paths are applicable to both setter and field injection.

The @Resource annotation is part of the [JSR-250](https://jcp.org/en/jsr/detail?id=250) annotation collection and is packaged with Jakarta EE. This annotation has the following execution paths, listed by precedence:

1. Match by Name
2. Match by Type
3. Match by Qualifier

These execution paths are applicable to both setter and field injection.

## What is autodetect?

Spring can automatically detect stereotyped classes and register corresponding BeanDefinitions

with the ApplicationContext. For example, the following two classes are eligible for such

autodetection:

@Service

**public class** SimpleMovieLister {

**private** MovieFinder movieFinder;

@Autowired

**public** SimpleMovieLister(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

}

@Repository

**public class** JpaMovieFinder **implements** MovieFinder {

*// implementation elided for clarity*

}

To autodetect these classes and register the corresponding beans, you need to add @ComponentScan

to your @Configuration class, where the basePackages attribute is a common parent package for

the two classes. (Alternatively, you can specify a comma/semicolon/space-separated list that includes

the parent package of each class.)

@Configuration

@ComponentScan(basePackages = "org.example")

**public class** AppConfig {

...

}

**Note**

for concision, the above may have used the value attribute of the annotation, i.e.

@ComponentScan("org.example")

The following is an alternative using XML

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

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

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

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

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

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

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

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

**<context:component-scan base-package**=**"org.example"/>**

**</beans>**

By default, classes annotated with @Component, @Repository, @Service, @Controller, or

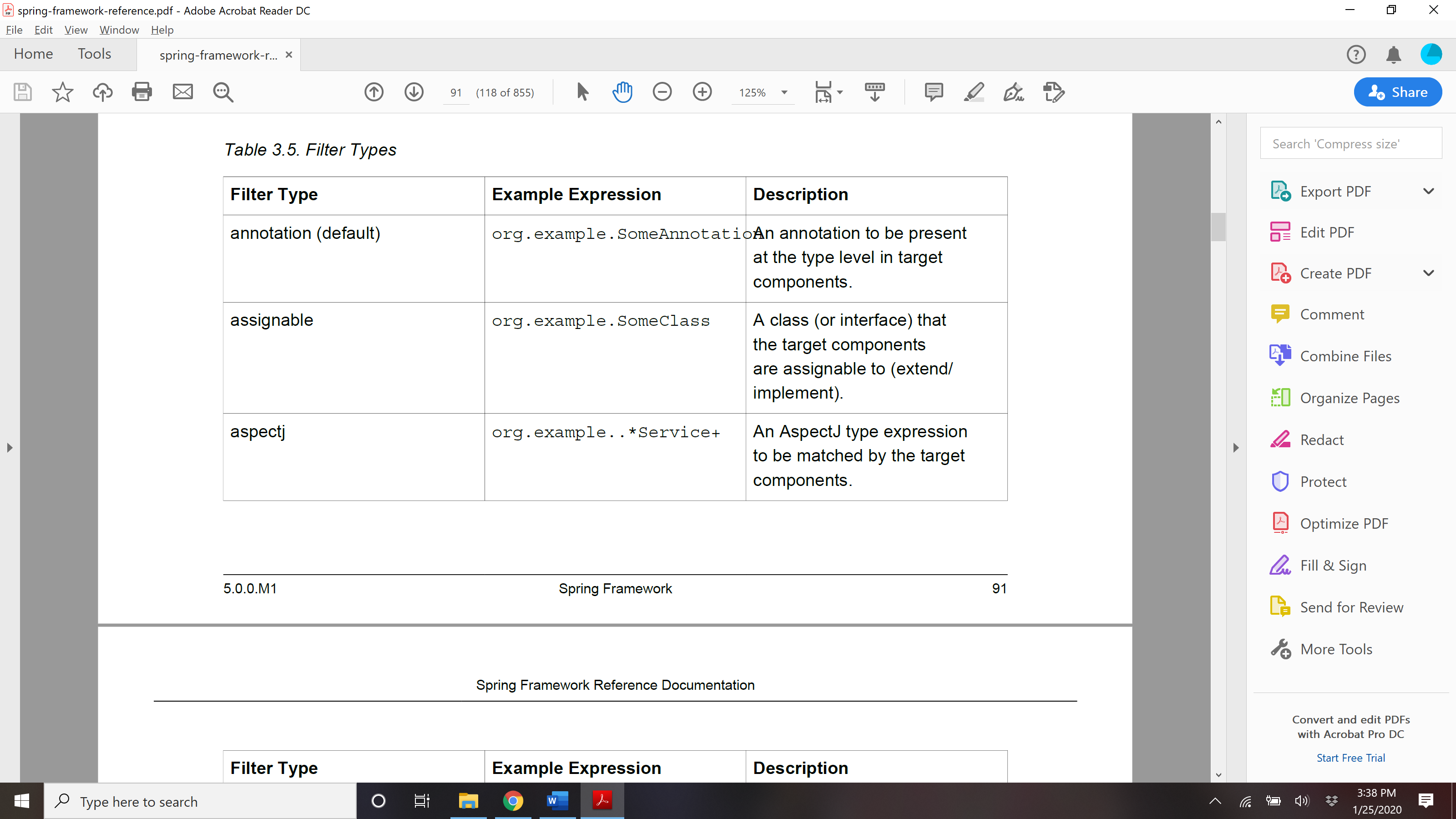
a custom annotation that itself is annotated with @Component are the only detected candidate

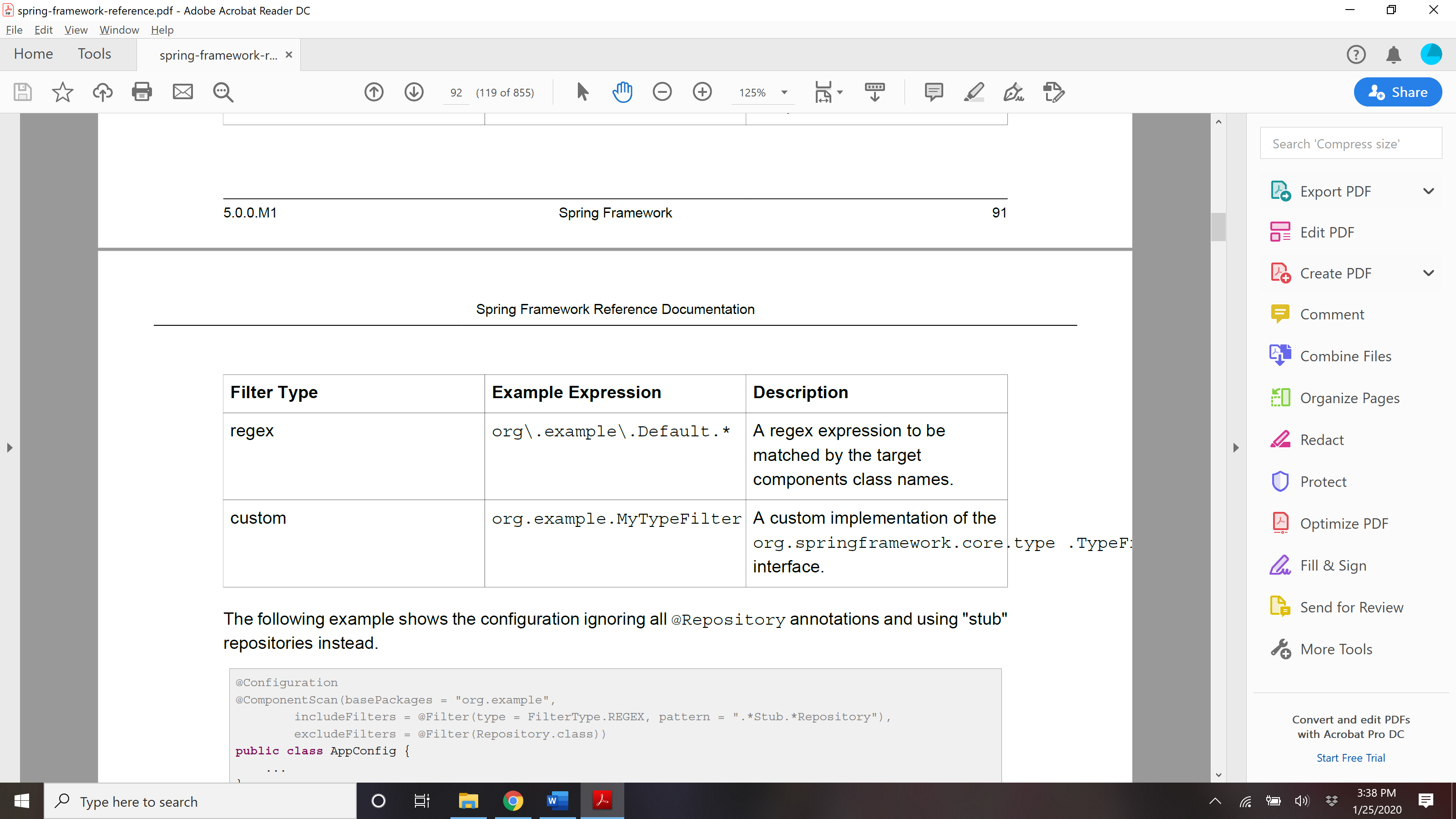
components. However, you can modify and extend this behavior simply by applying custom filters. Add

them as *includeFilters* or *excludeFilters* parameters of the @ComponentScan annotation (or as *includefilter*

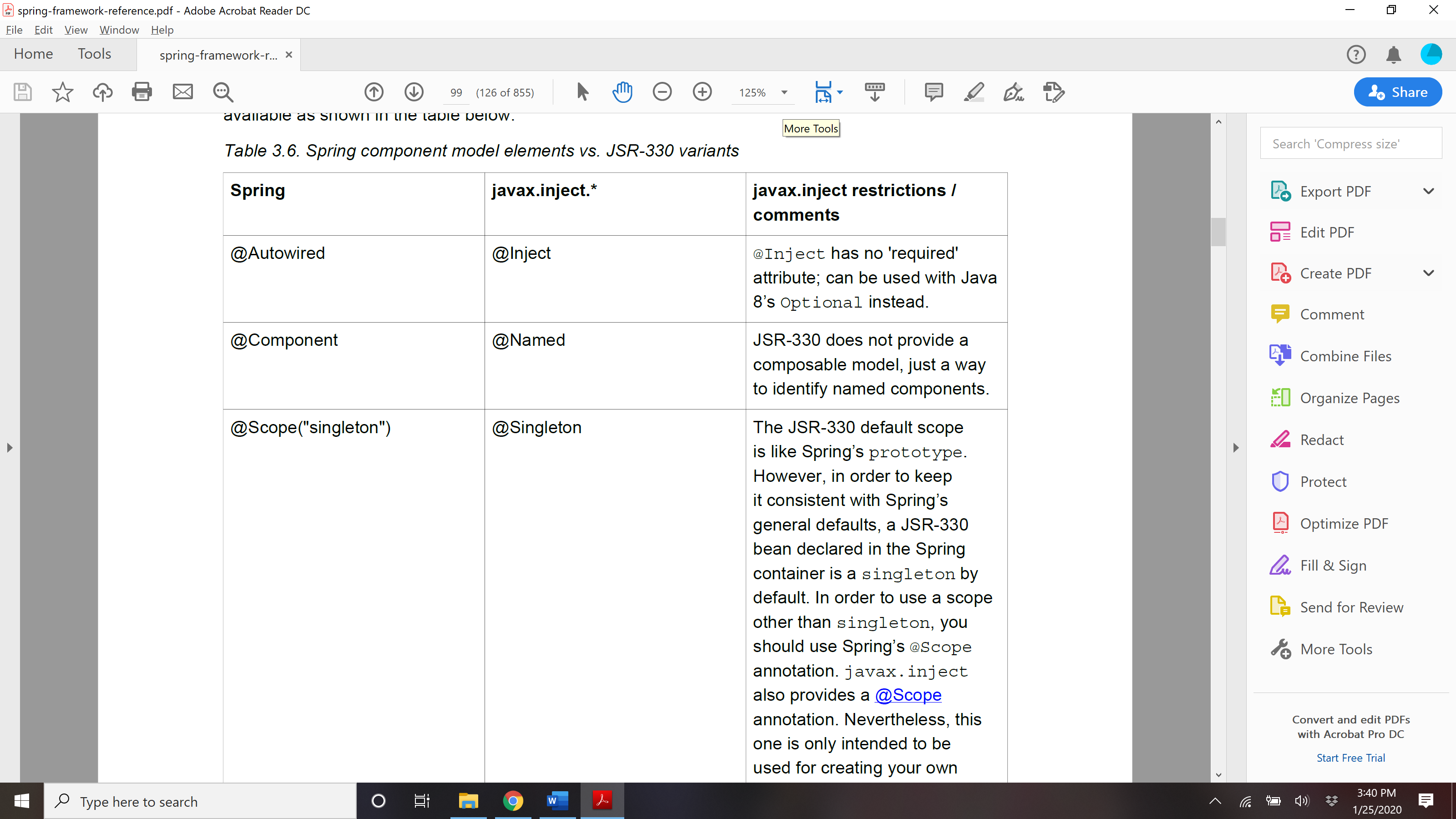
or *exclude-filter* sub-elements of the component-scan element). Each filter element requires the

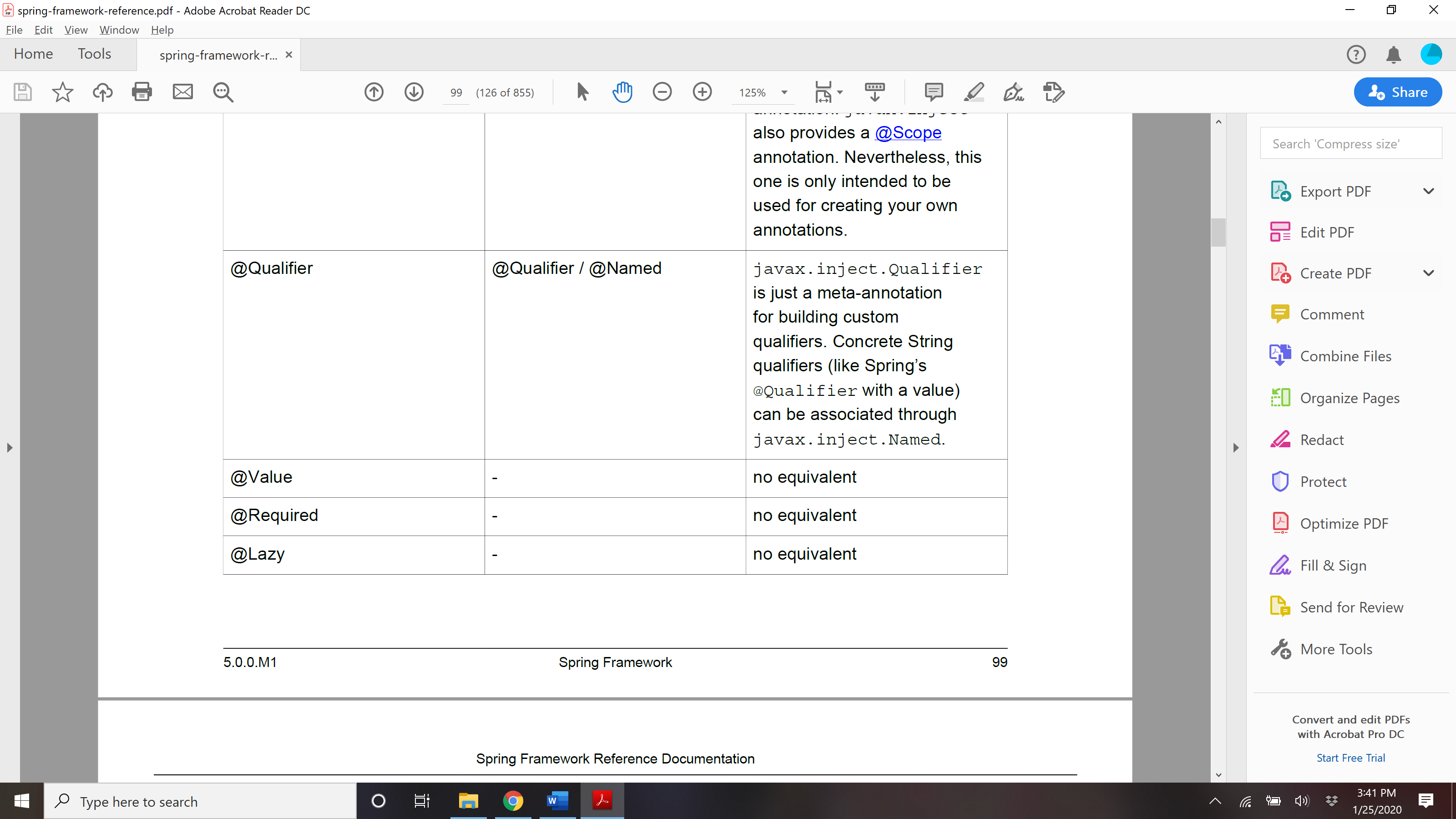
type and expression attributes. The following table describes the filtering options.





## **@Named: a standard equivalent to the @Component annotation**





## @Bean and @ Configuration?

The central artifacts in Spring’s new Java-configuration support are @Configuration-annotated

classes and @Bean-annotated methods.

The @Bean annotation is used to indicate that a method instantiates, configures and initializes a

new object to be managed by the Spring IoC container. For those familiar with Spring’s <beans/>

XML configuration the @Bean annotation plays the same role as the <bean/> element. You can use

@Bean annotated methods with any Spring @Component, however, they are most often used with

@Configuration beans.

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean

definitions. Furthermore, @Configuration classes allow inter-bean dependencies to be defined by

simply calling other @Bean methods in the same class. The simplest possible @Configuration class

would read as follows:

@Configuration

**public class** AppConfig {

@Bean

**public** MyService myService() {

**return new** MyServiceImpl();

}

}

The AppConfig class above would be equivalent to the following Spring <beans/> XML:

**<beans>**

**<bean id**=**"myService" class**=**"com.acme.services.MyServiceImpl"/>**

**</beans>**

**Full @Configuration vs 'lite' @Beans mode?**

When @Bean methods are declared within classes that are *not* annotated with @Configuration

they are referred to as being processed in a 'lite' mode. For example, bean methods declared in

a @Component or even in a *plain old class* will be considered 'lite'.

Unlike full @Configuration, lite @Bean methods cannot easily declare inter-bean dependencies.

Usually one @Bean method should not invoke another @Bean method when operating in 'lite' mode.

Only using @Bean methods within @Configuration classes is a recommended approach of

ensuring that 'full' mode is always used. This will prevent the same @Bean method from accidentally

being invoked multiple times and helps to reduce subtle bugs that can be hard to track down when

operating in 'lite' mode

## AnnotationConfigApplicationContext

Used for instantiating ApplicationContext for java based/annotation based files.

**public static void** main(String[] args) {

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

MyService myService = ctx.getBean(MyService.**class**);

myService.doStuff();

}

**public static void** main(String[] args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(MyServiceImpl.**class**,

Dependency1.**class**, Dependency2.**class**);

MyService myService = ctx.getBean(MyService.**class**);

myService.doStuff();

}

**public static void** main(String[] args) {

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(AppConfig.**class**, OtherConfig.**class**);

ctx.register(AdditionalConfig.**class**);

ctx.refresh();

MyService myService = ctx.getBean(MyService.**class**);

myService.doStuff();

}

## Component scan

To enable component scanning, just annotate your @Configuration class as follows:

@Configuration

@ComponentScan(basePackages = "com.acme")

**public class** AppConfig {

...

}

**Tip**

Experienced Spring users will be familiar with the XML declaration equivalent from Spring’s

context: namespace

**<beans>**

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

**</beans>**

## Annotation for @Bean

@Bean(name =xx, init-method = yy, destroy-method=zz)

@Scope

@Descrciption()

@Lazy

Public void getMyBean(){};

## Import in java based config?

@Configuration

**public class** ConfigA {

@Bean

**public** A a() {

**return new** A();

}

}

@Configuration

@Import(ConfigA.class)

**public class** ConfigB {

@Bean

**public** B b() {

**return new** B();

}

}

## @Conditional

## @Profile

The @Profile annotation allows you to indicate that a component is eligible for registration when

one or more specified profiles are active. Using our example above, we can rewrite the dataSource

configuration as follows:

@Configuration

**@Profile("dev")**

**public class** StandaloneDataConfig {

@Bean

**public** DataSource dataSource() {

**return new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript(***"classpath:com/bank/config/sql/schema.sql"***)

.addScript(***"classpath:com/bank/config/sql/test-data.sql"***)

.build();

}

}

@Configuration

**@Profile("production")**

**public class** JndiDataConfig {

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup(***"java:comp/env/jdbc/datasource"***);

}

}

If a @Configuration class is marked with @Profile, all of the @Bean methods and @Import

annotations associated with that class will be bypassed unless one or more of the specified profiles

are active. If a @Component or @Configuration class is marked with @Profile({"p1",

"p2"}), that class will not be registered/processed unless profiles 'p1' and/or 'p2' have been

activated. If a given profile is prefixed with the NOT operator (!), the annotated element will

be registered if the profile is **not** active. For example, given @Profile({"p1", "!p2"}),

registration will occur if profile 'p1' is active or if profile 'p2' is not active.

**Activating a profile:**

**Via Program:**

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles(***"dev"***);

ctx.register(SomeConfig.**class**, StandaloneDataConfig.**class**, JndiDataConfig.**class**);

ctx.refresh();

OR

-Dspring.profiles.active=***"profile1,profile2"***

***OR***

***In Sring boot -*** spring.profiles.active=dev in application.properties

<https://www.baeldung.com/spring-profiles>

## @PropertyResources

A way to define external property file

@Configuration

**@PropertySource("classpath:/com/myco/app.properties")**

**public class** AppConfig {

@Autowired

Environment env;

@Bean

**public** TestBean testBean() {

TestBean testBean = **new** TestBean();

testBean.setName(env.getProperty(***"testbean.name"***));

**return** testBean;

}

}

## Additional capability of Application context?

*Access to messages in i18n-style*, through the MessageSource interface.

• *Access to resources*, such as URLs and files, through the ResourceLoader interface.

• *Event publication* to namely beans implementing the ApplicationListener interface, through the

use of the ApplicationEventPublisher interface.

• *Loading of multiple (hierarchical) contexts*, allowing each to be focused on one particular layer, such

as the web layer of an application, through the HierarchicalBeanFactory interface.

**BeanFactory or ApplicationContext?**

Use an ApplicationContext unless you have a good reason for not doing so.

Because the ApplicationContext includes all functionality of the BeanFactory, it is generally

recommended over the BeanFactory, except for a few situations such as in embedded applications

running on resource-constrained devices where memory consumption might be critical and a few

extra kilobytes might make a difference. However, for most typical enterprise applications and

systems, the ApplicationContext is what you will want to use. Spring makes *heavy* use of

the BeanPostProcessor extension point (to effect proxying and so on). If you use only a plain

BeanFactory, a fair amount of support such as transactions and AOP will not take effect, at least not

without some extra steps on your part. This situation could be confusing because nothing is actually

wrong with the configuration.

The following table lists features provided by the BeanFactory and ApplicationContext interfaces

and implementations.

