Core Spring 5 Certification in Detail

Ivan Krizsan

Version: 2019-10-19

Introduction

This book contains a walk-through of the Core Spring 5 certification topics.
The text is deliberately kept brief. Please consult the references for in-depth discussions.

My recommendation to those aspiring to gain thorough knowledge of the topics covered by the Core Spring certification is to try to answer each and every topic and before reading this document.

Organization

This book is organized according to the Pivotal Core Spring Certification Study Guide available at the time of writing.
There are eight major sections:

- Container, Dependency and IoC
- Aspect Oriented Programming
- Data Management
- Spring Data JPA
- Spring MVC and the Web Layer
- Security
- REST
- Testing
- Spring Boot Intro
- Spring Boot Auto Configuration
- Spring Boot Actuator
- Spring Boot Testing

Table of Contents

Chapter 1

Container, Dependency and IoC

What is dependency injection and what are the advantages?

Dependency injection is the process whereby a framework or such, for example the Spring framework, establishes the relationships between different parts of an application. This as opposed to the application code itself being responsible of establishing these relationships.
When using the Spring framework for Java development, some of the advantages of dependency injection are:

- Reduced coupling between the parts of an application.
- Increased cohesion of the parts of an application.
- Increased testability.
- Better design of applications when using dependency injection.
- Increased reusability.
- Increased maintainability.
- Standardizes parts of application development.
- Reduces boilerplate code.
  No code needs to be written to establish relationships in domain classes. Such code or configuration is separated into XML or Java configuration classes.

References:

- Spring 5 Reference: Introduction to the Spring IoC container and beans
- Wikipedia: Dependency injection
- Wikipedia: Inversion of control

What is an interface and what are the advantages of making use of them in Java?

A Java (Java 8 and later) interface is a reference type that can contain the following:

- Constants
- Method signatures
  These are methods that have no implementation.
- Default methods
  A method with an implementation that, if not implemented in a class that implements the interface, will be used as a default implementation of the method in question. This can be useful when adding new method(s) to an interface and not wanting to modify all the classes that implement the interface.
- Static methods
  Static methods with implementation.
- Nested types
  Such a nested type can be an enumeration.

Some advantages of using interfaces in Java programming are:

- Allows for decoupling of a contract and its implementation(s).
  The contract is the interface and the implementations are the classes that implement the interface. This allows for implementations to be easily interchanged.
  Interfaces can be defined separately from the implementations.
- Allows for modularization of Java programs.
- Allowing for handling of groups of object in a similar fashion.
  For example, all objects of classes implementing the *java.util.List* Java interface can be used in the same way.
- Increase testability.
  Using interface types when referencing other objects make it easy to replace such references with mock or stub objects that implement the same interface(s).

Why are interfaces recommended for Spring beans?

The following are reasons why it is recommended to define interfaces that are later implemented by the classes implement the Spring beans in an application:

- Increased testability.
  Beans can be replaced with mock or stub objects that implement the same interface(s) as the real bean implementation.
- Allows for use of the JDK dynamic proxying mechanism.
- Allows for easier switching of Spring bean implementation.
- Allows for hiding implementation.
  For instance, a service implemented in a module only have a public interface while the implementation is only visible within the module.
  Hiding the implementation also allows the developer to freely refactor code to methods, without having to fear that such methods will be visible outside of the module containing the implementation.

References:

- Oracle: The Java Tutorials – What is an interface?
- Wikipedia: Interface (Java)
- Oracle: The Java Tutorial – Interfaces
- Oracle: Dynamic Proxy Classes

What is meant by application-context?

The application context in a Spring application is a Java object that implements the *ApplicationContext* interface and is responsible for:

- Instantiating beans in the application context.
- Configuring the beans in the application context.
- Assembling the beans in the application context.
- Managing the life-cycle of Spring beans.
  There are exceptions to this which will be discussed later.

Given the interfaces the *ApplicationContext* interface inherits from, an application context have the following properties:

- Is a bean factory.
  A bean factory instantiates, configures and assembles Spring beans. Configuration and assembly is value and dependency injection. A bean factory also manages the beans.
- It is a hierarchical bean factory.
  That is a bean factory that can be part of a hierarchy of bean factories.
- Is a resource loader that can load file resources in a generic fashion.
- It is an event publisher.
  As such it publishes application events to listeners in the application.
- It is a message source.
  Can resolve messages and supports internationalization.
- Is an environment.
  From such an environment, properties can be resolved. The environment also allows maintaining named groups of beans, so-called profiles. The beans belonging to a certain profile are registered with the application context only when the profile is active.

There can be more than one application context in a single Spring application. Multiple application contexts can be arranged in a parent-child hierarchy where the relation is directional from child context to parent context. Many child contexts can have one and the same parent context.

Some commonly used implementations of the *ApplicationContext* interface are:

- AnnotationConfigApplicationContext
  Standalone application context used with configuration in the form of annotated classes.
- AnnotationConfigWebApplicationContext
  Same as *AnnotationConfigApplicationContext* but for web applications.
- ClassPathXmlApplicationContext
  Standalone application context used with XML configuration located on the classpath of the application.
- FileSystemXmlApplicationContext
  Standalone application context used with XML configuration located as one or more files in the file system.

- XmlWebApplicationContext
  Web application context used with XML configuration.

References:

- Spring 5 Reference: Introduction to the Spring IoC Container and Beans
- Spring 5 Reference: Container Overview
- Spring 5 Reference: Additional Capabilities of the ApplicationContext
- Spring 5 API: ApplicationContext

How are you going to create a new instance of an ApplicationContext?

The way to create an application context differs depending on the type of application context that is to be created. Below follows examples for some common types of application contexts.

An application context implementation is chosen depending on what type of configuration is more commonly used. It is possible to mix the two types of configuration.

Non-Web Applications

For non-web applications, creating an application context only involves creating a new instance of the appropriate class implementing the *ApplicationContext* interface.

AnnotationConfigApplicationContext

This type of application context is used with standalone applications that uses Java configuration, that is classes annotated with @Configuration.

There are four constructors in *AnnotationConfigApplicationContext* with these signatures:

```
/* Creates an empty application context. */
public AnnotationConfigApplicationContext()
```

```
/*
 * Creates an application context that uses the supplied bean factory.
 * Whether the application context is empty depends on the supplied bean factory.
 */
public AnnotationConfigApplicationContext(DefaultListableBeanFactory beanFactory)
```

```
/*
 * Creates an application context populated with the beans from the supplied
 * classes annotated with @Configuration.
 */
public AnnotationConfigApplicationContext(Class<?>... annotatedClasses)
```

```
/*
 * Creates an application context populated with the beans from the
 * classes annotated with @Configuration found in the supplied package and its
 * sub-packages.
 */
public AnnotationConfigApplicationContext(String... basePackages)
```

This example shows how to create an instance of this application context that will be populated with the beans from a Java class annotated with the @Configuration annotation:

```
AnnotationConfigApplicationContext theApplicationContext =
            new AnnotationConfigApplicationContext(MyConfiguration.class);
```

This example shows how to create and instance of this application context that will be populated with the beans from all Java classes annotated with the @Configuration annotation in a certain package, including configuration found in any sub-packages:

```
AnnotationConfigApplicationContext theParentApplicationContext =
            new AnnotationConfigApplicationContext(
                "se.ivankrizsan.spring.examples.configuration");
```

Web Applications

For web applications, creating an application context is slightly more involved.

Servlet 2 – ContextLoaderListener and web.xml

With the Servlet 2 standard, the minimum required configuration to create an application context is a web.xml file located in WEB-INF and a Spring XML configuration file, also located in WEB-INF. The web.xml file typically looks something like this:

```
<web-app id="WebApp_ID" version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>SpringWebServlet2</display-name>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/springwebservlet2-config.xml</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
```

The Spring *ContextLoaderListener* creates the root Spring web application context of a web application. This does load a Spring web application root context, but there is no way of interacting with the web application through a browser since there is no dispatcher servlet to receive and route requests.

Servlet 2 – ContextLoaderListener, DispatcherServlet and web.xml

In order to be able to receive requests in a Servlet 2 based Spring web application, a *DispatcherServlet* servlet also need to be configured in the web.xml file:

```
<web-app id="WebApp_ID" version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>SpringWebServlet2</display-name>

    <context-param>
```

```
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/springwebservlet2-config.xml</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>SpringDispatcherServlet</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>SpringDispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

The dispatcher servlet will read a Spring XML configuration file named [dispatcher servlet name]-servlet.xml located in the WEB-INF directory and create a Spring application context that is a child to the Spring application root context created by the *ContextLoaderlistener*.
In the above example, the file is to be named SpringDispatcherServlet-servlet.xml. In this Spring configuration file Spring MVC controllers can be configured.
The contextConfigLocation context parameter can be empty if no Spring application context for the servlet is required. In such a case there will be only a root Spring application context in the web application. It is possible to run a Spring web application with only a root application context.

Servlet 3 – Web Application Initializers

In the Servlet 3 standard, the web.xml file has become optional and a class implementing the *WebApplicationInitializer* can be used to create a Spring application context. The following classes implement the *WebApplicationInitializer* interface:

- AbstractContextLoaderInitializer
  Abstract base class that registers a *ContextLoaderListener* in the servlet context.
- AbstractDispatcherServletInitializer
  Abstract base class that registers a *DispatcherServlet* in the servlet context.

- AbstractAnnotationConfigDispatcherServletInitializer
  Abstract base class that registers a *DispatcherServlet* in the servlet context and uses Java-based Spring configuration.
- AbstractReactiveWebInitializer
  Creates a Spring application context that uses Java-based Spring configuration. Creates a Spring reactive web application in the servlet container.

Servlet 3 – XmlWebApplicationContext

As with non-web application the Spring application context can be configured using either XML configuration file(s) or Java configuration (class(es) annotated with the @Configuration annotation).

XmlWebApplicationContext has one single constructor that takes no parameters. The following example shows how a *WebApplicationInitializer* that creates and configures an instance of *XmlWebApplicationContext* that in turn reads Spring bean configuration from an XML file.

```
public class MyXmlWebApplicationInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(final ServletContext inServletContext) {
        /* Create Spring web application context using XML configuration. */
        final XmlWebApplicationContext theXmlWebApplicationContext =
            new XmlWebApplicationContext();
        theXmlWebApplicationContext
            .setConfigLocation("/WEB-INF/applicationContext.xml");
        theXmlWebApplicationContext.setServletContext(inServletContext);
        theXmlWebApplicationContext.refresh();
        theXmlWebApplicationContext.start();

        /*
         * Create and register the DispatcherServlet.
         * This is not strictly necessary if the application does not need
         * to receive web requests.
         */
        final DispatcherServlet theDispatcherServlet =
            new DispatcherServlet(theXmlWebApplicationContext);
        ServletRegistration.Dynamic theServletRegistration =
            inServletContext.addServlet("app", theDispatcherServlet);
        theServletRegistration.setLoadOnStartup(1);
        theServletRegistration.addMapping("/app/*");
    }
}
```

Note that a servlet context must be set on the web application context.

Servlet 3 – AnnotationConfigWebApplicationContext

AnnotationConfigWebApplicationContext has only one single constructor taking no parameters. With this application context, configuration is registered after creation of the context, as can be seen in this example:

```
public class MyJavaConfigWebApplicationInitializer implements WebApplicationInitializer
{

    @Override

    public void onStartup(ServletContext inServletContext) {

        /* Create Spring web application context using Java configuration. */

        final AnnotationConfigWebApplicationContext
            theAnnotationConfigWebApplicationContext =
                new AnnotationConfigWebApplicationContext();

        theAnnotationConfigWebApplicationContext.setServletContext(inServletContext);

        theAnnotationConfigWebApplicationContext.register(
            ApplicationConfiguration.class);

        theAnnotationConfigWebApplicationContext.refresh();


        /* Create and register the DispatcherServlet. */

        final DispatcherServlet theDispatcherServlet =

            new DispatcherServlet(theAnnotationConfigWebApplicationContext);

        ServletRegistration.Dynamic theServletRegistration =

            inServletContext.addServlet("app", theDispatcherServlet);

        theServletRegistration.setLoadOnStartup(1);

        theServletRegistration.addMapping("/app-java/*");

    }

}
```

References:

- Spring 5 Reference: DispatcherServlet

Can you describe the lifecycle of a Spring Bean in an ApplicationContext?

The lifecycle of a Spring bean looks like this:

- Spring bean configuration is read and metadata in the form of a *BeanDefinition* object is created for each bean.
- All instances of *BeanFactoryPostProcessor* are invoked in sequence and are allowed an opportunity to alter the bean metadata.
- For each bean in the container:
  - An instance of the bean is created using the bean metadata.
  - Properties and dependencies of the bean are set.
  - Any instances of *BeanPostProcessor* are given a chance to process the new bean instance before and after initialization.
    - Any methods in the bean implementation class annotated with @PostConstruct are invoked.
      This processing is performed by a *BeanPostProcessor*.
    - Any *afterPropertiesSet* method in a bean implementation class implementing the *InitializingBean* interface is invoked.
      This processing is performed by a *BeanPostProcessor*. If the same initialization method has already been invoked, it will not be invoked again.
    - Any custom bean initialization method is invoked.
      Bean initialization methods can be specified either in the value of the *init-method* attribute in the corresponding <bean> element in a Spring XML configuration or in the *initMethod* property of the @Bean annotation.
      This processing is performed by a *BeanPostProcessor*. If the same initialization method has already been invoked, it will not be invoked again.
    - The bean is ready for use.
- When the Spring application context is to shut down, the beans in it will receive destruction callbacks in this order:
    - Any methods in the bean implementation class annotated with @PreDestroy are invoked.
    - Any *destroy* method in a bean implementation class implementing the *DisposableBean* interface is invoked.
      If the same destruction method has already been invoked, it will not be invoked again.

    - Any custom bean destruction method is invoked.
      Bean destruction methods can be specified either in the value of the destroy-method attribute in the corresponding <bean> element in a Spring XML configuration or in the *destroyMethod* property of the @Bean annotation.
      If the same destruction method has already been invoked, it will not be invoked again.

References:

- Spring 5 Reference: Bean overview
- Spring 5 Reference: Dependencies
- Spring 5 Reference: Customizing the nature of a bean
- Spring 5 Reference: Container Extension Points
- Spring 5 Reference: Receiving lifecycle callbacks

How are you going to create an ApplicationContext in an integration test?

Depending on whether JUnit 4 or JUnit 5 is used, the annotation @RunWith (JUnit 4) or @ExtendWith (JUnit 5) is used to annotate the test-class. In addition, the annotation @ContextConfiguration in both cases to specify either the XML configuration file(s) or the Java class(es) containing the Spring configuration to be loaded into the application context for the test.

JUnit 4 Example

The following code shows a JUnit 4 test that creates a Spring application context:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes=MyConfiguration.class)
public class JUnit4SpringTest {
    @Autowired
    protected MyBean mMyBean;
    @Autowired
    protected ApplicationContext mApplicationContext;

    @Test
    public void contextLoads() {
        final String theMessage = mMyBean.getMessage();
        System.out.println("Message from my bean is: " + theMessage);
        System.out.println("Application context: " + mApplicationContext);
    }
}
```

JUnit 5 Example

The following code shows a JUnit 5 test that creates a Spring application context:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;

/*
```

```
 * The @SpringJUnitConfig annotation is a combination of the JUnit 5
 * @ExtendWith(SpringExtension.class) annotation and the Spring
 * @ContextConfiguration annotation.
 */
@SpringJUnitConfig(classes=MyConfiguration.class)
public class JUnit5SpringTest {
      @Autowired
      protected MyBean mMyBean;
      @Autowired
     protected ApplicationContext mApplicationContext;

      @Test
      public void contextLoads() {
            final String theMessage = mMyBean.getMessage();
            System.out.println("Message from my bean is: " + theMessage);
        System.out.println("Application context: " + mApplicationContext);
      }
}
```

Web Application Context

In a test that loads a web application context, the web application context can be injected into the test class by using the @WebAppConfiguration annotation as shown in this example:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.web.context.WebApplicationContext;

/*
 * The @WebAppConfiguration tells the Spring test runner that the Spring
 * application context to be created for the test is
 * of the type {@code WebApplicationContext}.
 */
@SpringJUnitConfig(classes=MyConfiguration.class)
@WebAppConfiguration
public class SpringWebJUnit5Test {
    @Autowired
    protected MyBean mMyBean;
    @Autowired
    protected WebApplicationContext mWebApplicationContext;

    @Test
```

```
    public void contextLoads() {
        final String theMessage = mMyBean.getMessage();
        System.out.println("Message from my bean is: " + theMessage);
        System.out.println("Web application context: " + mWebApplicationContext);
    }
}
```

References:

- Spring 5 Reference: Spring TestContext Framework

What is the preferred way to close an application context? Does Spring Boot do this for you?

The preferred way to close an application context depends on the type of application.

Standalone Application

In a standalone non-web Spring application, there are two ways by which the Spring application context can be closed.

- Registering a shutdown-hook by calling the method *registerShutdownHook*, also implemented in the *AbstractApplicationContext* class.
  This will cause the Spring application context to be closed when the Java virtual machine is shut down normally. This is the recommended way to close the application context in a non-web application.
- Calling the *close* method from the *AbstractApplicationContext* class.
  This will cause the Spring application context to be closed immediately.

Web Application

In a web application, closing of the Spring application context is taken care of by the *ContextLoaderListener*, which implements the *ServletContextListener* interface. The *ContextLoaderListener* will receive a *ServletContextEvent* when the web container stops the web application.

Spring Boot Closing Application Context

Spring Boot will register a shutdown-hook as described above when a Spring application that uses Spring Boot is started.
The mechanism described above with the *ContextLoaderListener* also applies to Spring Boot web applications.

References:

- Spring 5 Reference: Shutting down the Spring IoC container gracefully in non-web applications
- Spring 5 API Documentation: ContextLoaderListener
- Spring Boot 2 Reference: Application Exit

Dependency Injection, Component Scanning, Stereotypes and Bean Scopes

Dependency injection is a two-step process that consists of:

- A Spring bean define its dependencies.
  This can be accomplished through, for instance, constructor arguments, properties that are to be set on an instance of the bean, parameters to a factory method.
- The Spring container injects the dependencies when it creates an instance of the bean.

Regardless of whether using Spring Java configuration or Spring XML configuration, a Java class from which one or more Spring beans are to be created declare its dependencies using one, or a combination of, the following ways:

- Constructor arguments.
- Factory method arguments.
- Setter methods.

The two main ways of dependency injection are constructor-based dependency injection, to which factory method arguments also belong, and setter-based dependency injection.

Describe dependency injection using Java configuration

When using Java configuration, the bean class first define instance variables to hold the properties of the bean. In addition, the bean class is implemented to have either constructor arguments or setter methods that stores values or references in the different properties of the bean.

Describe dependency injection using annotations (@Component, @Autowired)

A class implementing a Spring bean can be annotated with Spring annotations, in order to facilitate dependency injection.

The @Component annotation and specialized annotations derived from the @Component annotation, such as @Service, @Repository, @Controller etc, allow Spring beans to be automatically detected at component scanning (see below). Thus the need to declare the bean in XML or Java configuration is eliminated.
These annotations are applied at class level, as shown in this example:

```
@Service
public class AdditionService {
    public long add(final long inFirstNumber, final long inSecondNumber) {
        return inFirstNumber + inSecondNumber;
    }
}
```

The @Autowired annotation can be applied to constructors, methods, parameters and properties of a class. The Spring container will attempt to satisfy dependencies by inspecting the contents of the application context and inject appropriate references or values.

Example:

```
@Service
public class CalculatorService {
    @Autowired
    protected AdditionService mAdditionService;
    @Autowired
    protected SubtractionService mSubtractionService;

    /* Insert implementation of the service here. */
}
```

If a bean class contains one single constructor, then annotating it with @Autowired is not required in order for the Spring container to be able to autowire dependencies. If a bean class contains more than one constructor and autowiring is desired, at least one of the constructors need to be annotated with @Autowired in order to give the container a hint on which constructor to use.

Describe component scanning and Stereotypes

Component scanning

Component, or classpath, scanning is the process using which the Spring container searches the classpath for classes annotated with stereotype annotations and registers bean definitions in the Spring container for such classes.

To enable component scanning, annotate a configuration class in your Spring application with the @ComponentScan annotation. The default component scanning behavior is to detect classes annotated with @Component or an annotation that itself is annotated with @Component. Note that the @Configuration annotation is annotated with the @Component annotation and thus are Spring Java configuration classes also candidates for auto-detection using component scanning.
Filtering configuration can be added to the @ComponentScan annotation as to include or exclude certain classes.

```
@ComponentScan(
    basePackages = "se.ivankrizsan.spring.corespringlab.service",
    basePackageClasses = CalculatorService.class,
    excludeFilters = @ComponentScan.Filter(type = FilterType.REGEX, pattern =
".*Repository"),
    includeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION, classes =
MyService.class)
)
```

```
public class CoreSpringLabApplication {
...
```

The above example configures component scanning:

- Using the basePackages property, the base package to be scanned is set to se.ivankrizsan.spring.corespringlab.services.
- Using the basePackageClasses property, a base package to be scanned is specified by setting a class located in the base package.
  This method of specifying a base package is preferred over using the basePackages property, due to better support from refactoring tooling.
- The excludeFilters property specifies a filter which selects classes for which no Spring bean definitions are to be registered.
  In this example, a regular expression which selects classes with name that ends with "Repository" to be excluded.
- The includeFilters specifies a filter which selects classes for which Spring bean definitions are to be registered.
  In this example, the filter selects classes annotated with the @MyService annotation.

Stereotype Annotations

Stereotype annotations are annotations that are applied classes that contains information of which role Spring beans implemented by the class fulfills. Stereotype annotations defined by Spring are:

Stereotype Annotation	Description
@Component	Root stereotype annotation that indicates that a class is a candidate for autodetection.
@Controller	Indicates that a class is a web controller.
@RestController	Indicates that a class is a specialized web controller for a REST service. Combines the @Controller and @ResponseBody annotations.
@Repository	Indicates that a class is a repository (persistence).
@Service	Indicates that a class is a service.
@Configuration	Indicates that a class contains Spring Java configuration (methods annotated with @Bean).

Describe scopes for Spring beans? What is the default scope?

The following default bean scopes exist in Spring 5.

Scope	Description
singleton	Single bean instance per Spring container.
prototype	Each time a bean is requested, a new instance is created.
request	Single bean instance per HTTP request. Only in web-aware Spring application contexts.
session	Single bean instance per HTTP session. Only in web-aware Spring application contexts.
application	Single bean instance per *ServletContext*. Only in web-aware Spring application contexts.
websocket	Single bean instance per *WebSocket*. Only in web-aware Spring application contexts.

The singleton scope is always the default bean scope.

References:

- Spring 5 Reference: Dependency Injection
- Spring 5 Reference: Bean scopes
- Spring 5 Reference: Java-based container configuration
- Spring 5 Reference: Autowiring collaborators
- Spring 5 Reference: Classpath scanning and managed components
- Spring 5 API Documentation: @ComponentScan
- Spring 5 API Documentation: @Component
- Spring 5 Reference: Combining Java and XML configuration

Are beans lazily or eagerly instantiated by default? How do you alter this behavior?

Singleton Spring beans in an application context are eagerly initialized by default, as the application context is created.
An instance of a prototype scoped bean is typically created lazily when requested. An exception is when a prototype scoped bean is a dependency of a singleton scoped bean, in which case the prototype scoped bean will be eagerly initialized.

To explicitly set whether beans are to be lazily or eagerly initialized, the @Lazy annotation can be applied either to:

- Methods annotated with the @Bean annotation.
  Bean will be lazy or not as specified by the boolean parameter to the @Lazy annotation (default value is true).
- Classes annotated with the @Configuration annotation.
  All beans declared in the configuration class will be lazy or not as specified by the boolean parameter to the @Lazy annotation (default value is true).
- Classes annotated with @Component or any related stereotype annotation.
  The bean created from the component class will be lazy or not as specified by the boolean parameter to the @Lazy annotation (default value is true).

References:

- Spring 5 Reference: Lazy-initialized beans
- Spring 5 Reference: Bean scopes
- Spring 5 API Reference: Lazy (annotation)

What is a property source? How would you use @PropertySource?

A property source in Spring's environment abstraction represents a source of key-value pairs. Examples of property sources are:

- The system properties of the JVM in which the Spring application is executed.
  As can be obtained by calling *System.getProperties()*.
- The system environment variables.
  As can be obtained by calling *System.getenv()*.
- Properties in a JNDI environment.
- Servlet configuration init parameters.
- Servlet context init parameters.
- Properties files.
  Both traditional properties file format and XML format are supported. See the *ResourcePropertySource* class for details.

The @PropertySource annotation can be used to add a property source to the Spring environment. The annotation is applied to classes annotated with @Configuration. Example:

```
@Configuration
@PropertySource("classpath:testproperties.properties")
public class TestConfiguration {
```

References:

- Spring 5 Reference: PropertySource abstraction
- Spring 5 Reference: @PropertySource
- Spring 5 API Documentation: Environment
- Spring 5 API Documentation: PropertyResolver
- Spring 5 API Documentation: PropertySource (class)
- Spring 5 API Documentation: ResourcePropertySource (class)
- Spring 5 API Documentation: PropertySource (annotation)

What is a BeanFactoryPostProcessor and what is it used for? When is it invoked?

BeanFactoryPostProcessor is an interface that defines the property (a single method) of a type of container extension point that is allowed to modify Spring bean meta-data prior to instantiation of the beans in a container. A bean factory post processor may not create instances of beans, only modify bean meta-data. A bean factory post processor is only applied to the meta-data of the beans in the same container in which it is defined in.

Examples of bean factory post processors are:

- *DeprecatedBeanWarner*
  Logs warnings about beans which implementation class is annotated with the @Deprecated annotation.
- *PropertySourcesPlaceholderConfigurer*
  Allows for injection of values from the current Spring environment the property sources of this environment. Typically values from the applications properties-file are injected using the @Value annotation.

The following example shows how to create a *PropertySourcesPlaceholderConfigurer* using Spring Java configuration:

```
@Bean
public static PropertySourcesPlaceholderConfigurer propertyConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
}
```

To influence the order in which bean factory post processors are invoked, their bean definition methods may be annotated with the @Order annotation. If you are implementing your own bean factory post processor, the implementation class can also implement the *Ordered* interface.

When is a bean factory post processor invoked?

The section above on the life-cycle of the IoC container describes at what point in the container's life-cycle bean factory post processors are invoked, see reference link below.

References:

- Spring 5 API Documentation: BeanFactoryPostProcessor
- Spring 5 Reference: Customizing configuration metadata with a BeanFactoryPostProcessor
- IoC Container Lifecycle

Why would you define a static @Bean method?

Static @Bean methods can be defined in order to create, for instance, a *BeanFactoryPostProcessor* that need to be instantiated prior to the instantiation of any beans that the *BeanFactoryPostProcessor* is supposed to modify before the beans are being used.

An example of such a *BeanFactoryPostProcessor* is the *PropertySourcesPlaceholderConfigurer* which function is discussed in detail in the next section.

References:

- Spring 5 API Documentation: Bean (annotation)

What is a ProperySourcesPlaceholderConfigurer used for?

PropertySourcesPlaceholderConfigurer is a *BeanFactoryPostProcessor* that resolves property placeholders, on the ${PROPERTY_NAME} format, in Spring bean properties and Spring bean properties annotated with the @Value annotation. When such a placeholder is encountered the corresponding value from the Spring environment and its property sources is injected into the property.

Using property placeholders, Spring bean configuration can be externalized into property files. This allows for changing for example the database server used by an application without having to rebuild and redeploy the application.

References:

- Spring 5 API Documentation: PropertySourcesPlaceholderConfigurer
- Spring 5 API Documentation: PropertySources
- Spring 5 API Documentation: PropertySource<T>

What is a BeanPostProcessor and how is it different to a BeanFactoryPostProcessor? What do they do? When are they called?

Both *BeanPostProcessor* and *BeanFactoryPostProcessor* are interfaces that allow for definition of container extensions. Examples of such container extensions are declarative transaction handling, Spring AOP, property placeholders, bean initialization and destruction methods.
Both *BeanPostProcessor* and *BeanFactoryPostProcessor* instances operate only on beans and bean definitions respectively that are defined in the same Spring container.

Classes implementing either the *BeanPostProcessor* or the *BeanFactoryPostProcessor* interfaces may also implement the *Ordered* interface in order to allow for setting a priority of a post-processor; a lower order value will cause the post-processor to be invoked earlier.

When defining a *BeanPostProcessor* or a *BeanFactoryPostProcessor* using an @Bean annotated method, it is recommended that the method is static, in order for the post-processor to be instantiated early in the Spring context creation process.

For a description of when the two types of post-processors are invoked, please refer to the section on the container life-cycle earlier.

BeanPostProcessor

A *BeanPostProcessor* is an interface that defines callback methods that allow for modification of bean instances. A *BeanPostProcessor* may even replace a bean instance with, for instance, an AOP proxy.
BeanPostProcessor-s can be registered programmatically using the *addBeanPostProcessor* method as defined in the *ConfigurableBeanFactory* interface.

Examples of *BeanPostProcessor*-s are:

- *AutowiredAnnotationBeanPostProcessor*
  Implements support for dependency injection with the @Autowired annotation.
- *PersistenceExceptionTranslationPostProcessor*
  Applies exception translation to Spring beans annotated with the @Repository annotation.

BeanFactoryPostProcessor

A *BeanFactoryPostProcessor* is an interface that defines callback methods that allow for implementation of code that modify bean definitions. Note that a *BeanFactoryPostProcessor* may interact with and modify bean definitions, bean metadata, but it may not instantiate beans.

Examples of *BeanFactoryPostProcessor*-s are:

- *PropertyOverrideConfigurer*
  Allows for overriding property values in Spring beans.

- PropertyPlaceholderConfigurer
  Allows for using property placeholders in Spring bean properties and replaces these with actual values, typically from a property file.

References:

- Spring 5 Reference: Container Extension Points
- Spring 5 API Documentation: BeanPostProcessor (interface)
- Spring 5 API Documentation: BeanFactoryPostProcessor (interface)
- Spring 5 API Documentation: Ordered (interface)

What is an initialization method and how is it declared on a Spring bean?

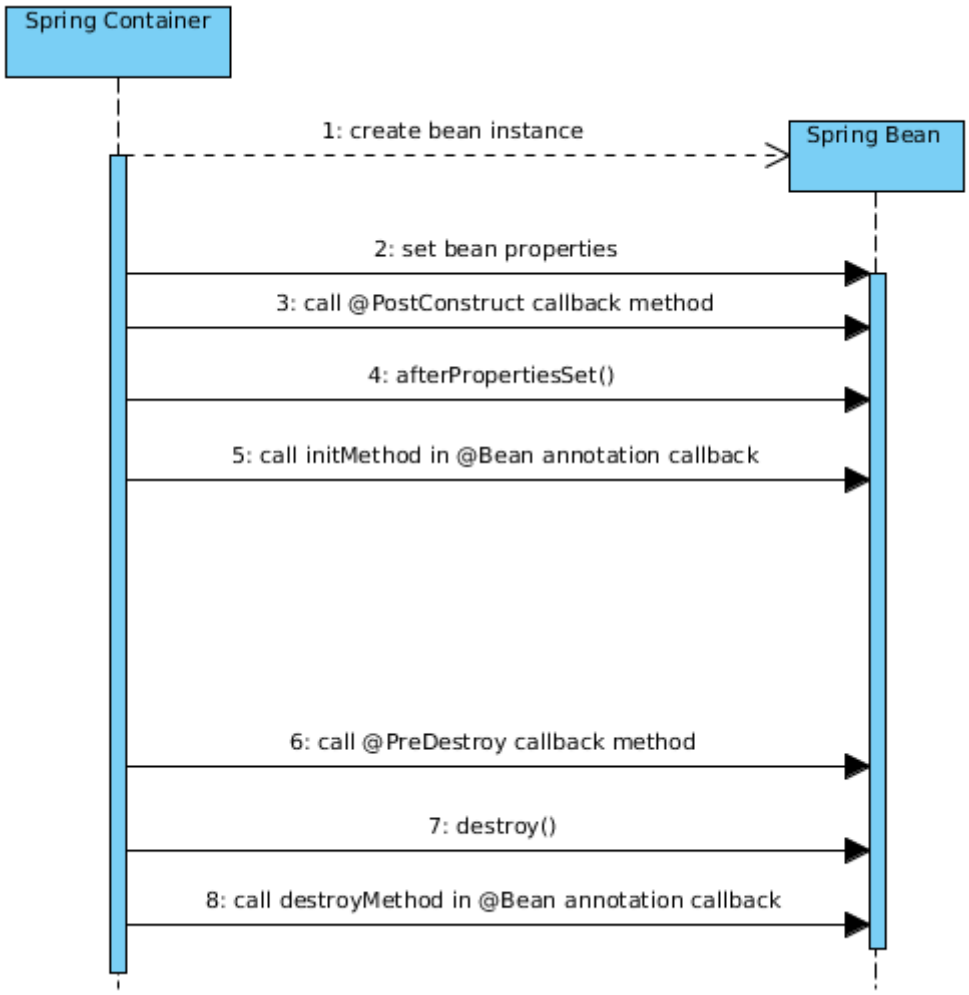
An initialization method is a method in a Spring bean that will be invoked by the Spring application container after all properties on the bean have been populated but before the bean is taken into use. An initialization method allow the Spring bean to perform any initialization that depend on the properties of the bean to have been set – such initialization cannot be performed in the constructor since the bean properties will not have been set when the constructor is being executed.

Excluding XML-based Spring bean configuration, there are three different ways to declare an initialization method:

- Implementing the *InitializingBean* interface and implementing the *afterPropertiesSet* method in the bean class.
  While heavily used in the Spring framework, this method is not recommended by the Spring Reference Documentation since it introduce unnecessary coupling to the Spring framework.
- Annotate the initialization method with the @PostConstruct annotation.
  @PostConstruct is a standard Java lifecycle annotation, as specified in JSR-250.
  An annotated method may have any visibility, may not take any parameters and may only have the void return type.
- Use the *initMethod* element of the @Bean annotation.

The above methods are invoked in the order shown in the sequence diagram below. Note that it is assumed that the callback methods in the bean have different names. If more than one initialization callback specify the same method then this method will only be invoked once and at the earliest available occasion given the ways used to specify the callback.
Example:
If there is a method named *initialize2* that is annotated with @PostConstruct and is also specified in the *initMethod* element in the corresponding @Bean annotation, then this method will only be invoked once immediately before any *afterPropertiesSet* method is invoked on the bean.

Ofcourse the *afterPropertiesSet* method will only be invoked if the bean implements the *InitializingBean* interface.



Ordering of initialization and destruction callbacks on a Spring bean.

Initialization methods are always called when a Spring bean is created, regardless of the scope of the bean.

References:

- Spring 5 Reference: Initialization Callbacks
- Spring 5 API Documentation: InitializingBean (interface)
- Spring 5 Reference: @PostConstruct and @PreDestroy
- JSR-250: Common Annotations for the Java Platform
- Spring 5 Reference: Receiving Lifecycle Callbacks
- Spring 5 Reference: Combining Lifecycle Mechanisms

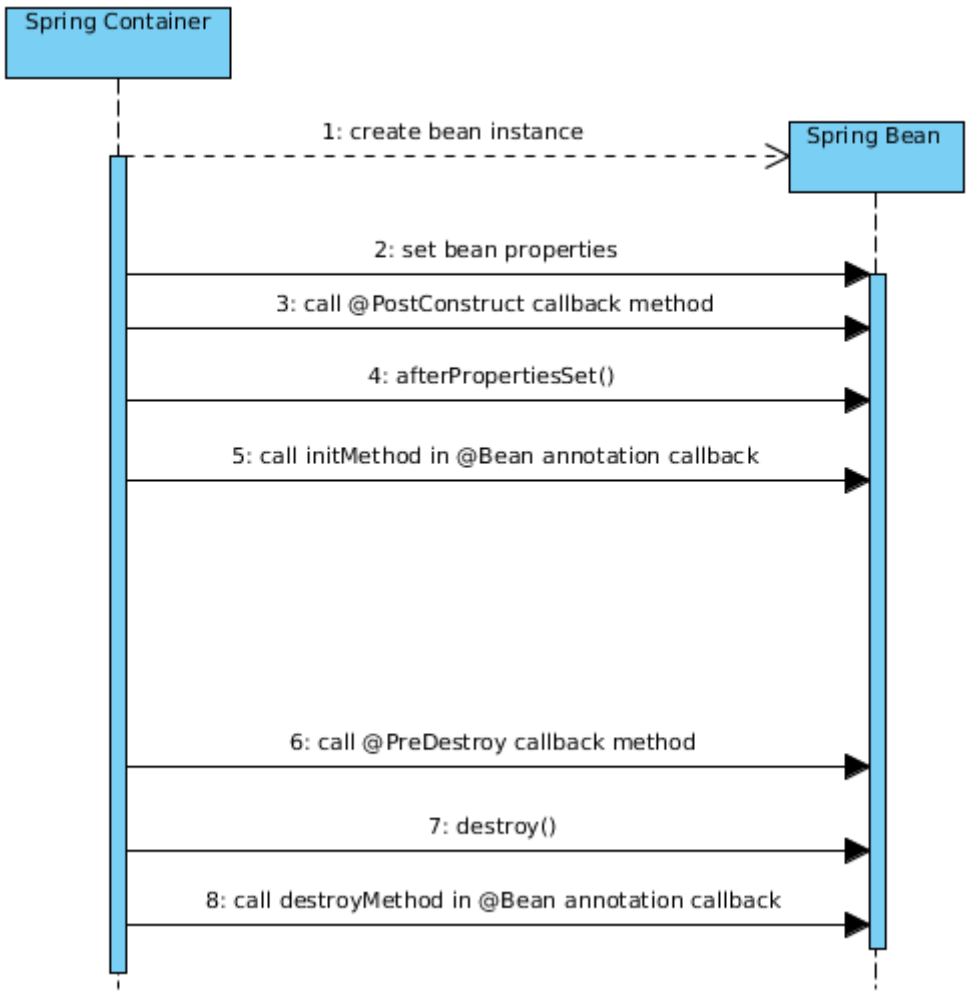
What is a destroy method, how is it declared and when is it called?

A destroy method is a method in a Spring bean that will be invoked by the Spring application container immediately before the bean is to be taken out of use, typically when the Spring application context is about to be closed. A destroy method allow the Spring bean to perform any cleanup or release any resources tied to the bean.

Excluding XML-based Spring bean configuration, there are three different ways to declare a destroy method:

- Implementing the *DisposableBean* interface and implementing the *destroy* method in the bean class.
  While heavily used in the Spring framework, this method is not recommended by the Spring Reference Documentation since it introduce unnecessary coupling to the Spring framework.
- Annotate the initialization method with the @PreDestroy annotation.
  @PreDestroy is a standard Java lifecycle annotation, as specified in JSR-250.
  An annotated method may have any visibility, may not take any parameters and may only have the void return type.
- Use the *destroyMethod* element of the @Bean annotation.

The above methods are invoked in the order shown in the sequence diagram below. Note that it is assumed that the callback methods in the bean have different names. If more than one destroy callback specify the same method then this method will only be invoked once and at the earliest available occasion given the ways used to specify the callback.



Order of initialization and destruction callbacks on a Spring bean.

When defining a Spring bean using Java configuration, methods named *close* and *shutdown* will automatically be registered as destruction callback methods by the Spring application container, as if these methods had been specified using the *destroyMethod* element of the @Bean annotation. To avoid this, set the *destroyMethod* element of the @Bean annotation to the empty string, like in this example:

```
@Bean(destroyMethod="")
public MyBeanClass myBeanWithACloseMethodNotToBeInvokedAsLifecycleCallback() {
    final MyBeanClass theBean = new MyBeanClass();
    return theBean;
}
```

With Spring beans that have prototype scope, that is for which a new bean instance will be created every time the Spring container receives a request for the bean, no destruction callback methods will be invoked by the Spring container.

References:

- Spring 5 API Documentation: DisposableBean (interface)

- Spring 5 Reference: @PostConstruct and @PreDestroy
- JSR-250: Common Annotations for the Java Platform
- Spring 5 Reference: Receiving Lifecycle Callbacks
- Spring 5 Reference: Combining Lifecycle Mechanisms

Consider how you enable JSR-250 annotations like @PostConstruct and @PreDestroy? When/how will they get called?

The *CommonAnnotationBeanPostProcessor* support, among other annotations, the @PostConstruct and @PreDestroy JSR-250 annotations.
When creating a Spring application context using an implementation that uses annotation-based configuration, for instance *AnnotationConfigApplicationContext*, a default *CommonAnnotationBeanPostProcessor* is automatically registered in the application context and no additional configuration is necessary to enable @PostConstruct and @PreDestroy.

For a discussion on when methods in a Spring bean annotated with @PostConstruct and @PreDestroy are called, please refer to the sections above on initialization and destroy methods.

References:

- Spring 5 Reference: Annotation-Based Container Configuration

How else can you define an initialization or destruction method for a Spring bean?

Please refer to the above sections on initialization and destruction methods.

What does component-scanning do?

Please refer to the section on Component Scanning earlier.

What is the behavior of the annotation @Autowired with regards to field injection, constructor injection and method injection?

Autowiring is a mechanism which enables more or less automatic dependency resolution primarily based on types. The basic procedure of dependency injection with the @Autowired annotation is as follows:

- The Spring container examines the type of the field or parameter that is to be dependency injected.
- The Spring container searches the application context for a bean which type matches the type of the field or parameter.
- If there are multiple matching bean candidates and one of them is annotated with @Primary, then this bean is selected and injected into the field or parameter.
- If there are multiple matching bean candidates and the field or parameter is annotated with the @Qualifier annotation, then the Spring container will attempt to use the information from the @Qualifier annotation to select a bean to inject.
  Please refer to the section below on the @Qualifier annotation.
- If there is no other resolution mechanism, such as the @Primary or @Qualifier annotations, and there are multiple matching beans, the Spring container will try to resolve the appropriate bean by trying to match the bean name to the name of the field or parameter. This is the default bean resolution mechanism used when autowiring dependencies.
- If still no unique match for the field or parameter can be determined, an exception will be thrown.

The following are common for all the different use-cases of the @Autowired annotation:

- Dependency injection, regardless of whether on fields, constructors or methods, is performed by the *AutowiredAnnotationBeanPostProcessor*.
  Due to this, the @Autowired annotation cannot be used in neither *BeanPostProcessor*-s nor in *BeanFactoryPostProcessor*-s.
- All dependencies annotated with @Autowired are required as default and an exception will be thrown if such a dependency cannot be resolved.
- If the type that is autowired is an array-type, then the Spring container will collect all beans matching the value-type of the array in an array and inject the array.
- If the type that is autowired is a collection type, then the Spring container will collect all beans matching the collection's value-type in a collection of the specified type and inject the collection.

- If the type that is autowired is a map with the key type being *String*, then the Spring container will collect all beans matching the value type of the map and insert these into the map with the bean name as key and inject the map.
- As per default, the order in which the matching beans injected in a array, collection or map are registered is the order in which they will appear in the array, collection or map.
  To affect the ordering, either have the bean classes implement the *Ordered* interface or annotate the @Bean methods with the @Order or @Priority annotations.
- Spring beans which type is a collection, including maps, can be autowired.
  Take care as to avoid conflicts with the above two means of injecting multiple beans at once, as they will take precedence over any collection-type beans you define yourself that has a value-type that is also used for one or more Spring beans.
- When autowiring arrays, collections or maps containing Spring beans, dependency injection will, as default, fail with an error if there are no matching beans.
  Thus an empty array, collection or map will not be injected. If the array, collection or map parameter is made optional, null (and not an empty array, collection or map) will be injected if there are no beans of the matching type.
- It is possible to autowire dependencies of the types *BeanFactory*, *ApplicationContext*, *Environment*, *ResourceLoader*, *ApplicationEventPublisher*, and *MessageSource* without having to declare beans of the corresponding type.
  These are objects

@Autowired and Field Injection

Fields of a Spring bean annotated with @Autowired can have any visibility and are injected after the bean instance has been created, before any initialization-methods are invoked.

@Autowired and Constructor Injection

Constructors in Spring bean classes can be annotated with the @Autowired annotation in order for the Spring container to look up Spring beans with the same types as the parameters of the constructor and supply these beans (as parameters) when creating an instance of the bean with the @Autowired-annotated constructor.

If there is only one single constructor with parameters in a Spring bean class, then there is no need to annotate this constructor with @Autowired – the Spring container will perform dependency injection anyway.

If there are multiple constructors in a Spring bean class and autowiring is desired, @Autowired may be applied to one of the constructors in the class. Only one single constructor may be annotated with @Autowired.

Constructors annotated with @Autowired does not have to be public in order for Spring to be able to create a bean instance of the class in question, but can have any visibility.

If a constructor is annotated with @Autowired, then all the parameters of the constructor are required. Individual parameters of such constructors can be declared using the Java 8 *Optional* container object, annotated with the @Nullable annotation or annotated with @Autowired(required=false) to indicate that the parameter is not required. Such parameters will be set to null, or *Optional.EMPTY* if the parameter is of the type *Optional*.

@Autowired and Method Injection

Methods can be annotated with @Autowired. Such methods can be:

- Regular setter-methods.
  These are methods which name starts with “set”, for example “setEnvironment”, that takes one single parameter and has the return type void.
- Methods with arbitrary names.
- Methods with more than one parameter.
- Methods with any visibility.
  Example: Setter-methods annotated with @Autowired can be private – the Spring container will still detect and invoke them.
- Methods that do not have a void return type.
  While possible, I see little reason to return anything from a method annotated with @Autowired.

If a method annotated with @Autowired(required = false) has multiple parameters then this method will not be invoked by the Spring container, and thus no dependency injection will take place, unless all the dependencies can be resolved in the Spring context.

If a method annotated with @Autowired, regardless of whether required is true or false, has parameters wrapped by the Java 8 *Optional*, then this method will always be invoked with the parameters for which dependencies can be resolved having a value wrapped in an *Optional* object. All parameters for which no dependencies can be resolved will have the value *Optional.EMPTY*.

References:

- Spring 5 Reference: Autowiring collaborators
- Spring 5 Reference: @Autowired
- Spring 5 API Documentation: @Autowired
- Spring 5 API Documentation: AutowiredAnnotationBeanPostProcessor
- Spring 5 API Documentation: @Nullable
- Java 8 API Documentation: Optional
- Spring 5 Reference: Fine-tuning annotation-based autowiring with qualifiers

What do you have to do, if you would like to inject something into a private field? How does this impact testing?

There are two basic cases of injecting a value or reference into a private field; either the source-code containing the private field to be injected can be modified or it cannot be modified. The latter case typically occurs with third-party libraries and generated code. Some of the following is only applicable in the case where the source-code can be modified.

Using @Autowired or @Value

Using the annotations @Autowired or @Value to annotate private fields in a class in order to perform injection of dependencies or values is perfectly feasible if the source-code can be modified. Both these annotations can also be applied to setter-methods. Dependency- or value-injection using setter-methods will work regardless of the visibility of the setter-method – that is, setter-methods may have any visibility, even private.

Using Constructor Parameters

Another alternative is to use constructor-parameter dependency injection, where the reference or value of the private field is assigned a value in the constructor using a constructor-parameter.

If not using annotations to have values or references injected into private fields of a Spring bean, then either constructor-parameter dependency injection or setter-methods must be used. If creating the bean in a Spring Java configuration class, then the constructor or the setter-method(s) must be visible from the Java configuration class. While the constructor or setter-method(s) can not have private visibility, package visibility may be used if the Java configuration class reside in the same package as the class implementing the Spring bean and if there is a wish to limit visibility.

Testing and Private Fields

When testing a class with private fields that are to have references injected into them that use @Autowired, setter-dependency injection or constructor-parameter dependency injection it is easy to replace the reference injected with a mock bean by using a test configuration that replaces the original bean.

To customize property values in a test, the @TestPropertySource annotation allows using either a test-specific property file or customizing individual property values.

If there is a need to inject a value or reference into a private field that do not have a public setter method, the Spring framework contains, among other testing utilities, the class *ReflectionTestUtils* which for example contain support for:

- Changing value of constants.
- Setting a value or reference into a non-public field.
- Invoking a non-public setter method.

- Invoking a non-public configuration or lifecycle callback method.

While *ReflectionTestUtils* do make it possible to access private properties, I personally feel it is preferable to develop your own code as to avoid private visibility and rather use package visibility, in which case fields etc (with package visibility) can be set or read from a test-class in the same package.

References:

- Spring 5 Reference: Unit Testing Support Classes
- Spring 5 API Documentation: @TestPropertySource

How does the @Qualifier annotation complement the use of @Autowired?

As described earlier, dependency injection with the @Autowired annotation works by matching the type of the field or parameter to be injected with the type of the Spring bean to be injected. The @Qualifier annotation adds additional control of the selection of the bean to inject if there are multiple beans of the same type in the Spring application context.

The @Qualifier annotation can be used at three different locations:

- At injection points.
- At bean definitions.
- At annotation definitions.
  This creates a custom qualifier annotation.

@Qualifier at Injection Points

As mentioned earlier, the @Qualifier annotation can aid in selecting one single bean to be dependency-injected into a field or parameter annotated with @Autowired when there are multiple candidates. The most basic use of the @Qualifier annotation is to specify the name of the Spring bean to be selected the bean to be dependency-injected.

@Qualifier at Bean Definitions

Qualifiers can also be applied on bean definitions by annotating a method annotated with @Bean in a configuration class with @Qualifier and supplying a value in the @Qualifier annotation. This will assign a qualifier to the bean and the same qualifier can later be used at an injection point to inject the bean in question.
If a bean has not been assigned a qualifier, the default qualifier, being the name of the bean, will be assigned the bean.

@Qualifier at Annotation Definitions

Annotation definitions can be annotated with the @Qualifier annotation in order to create custom qualifier annotations.

References:

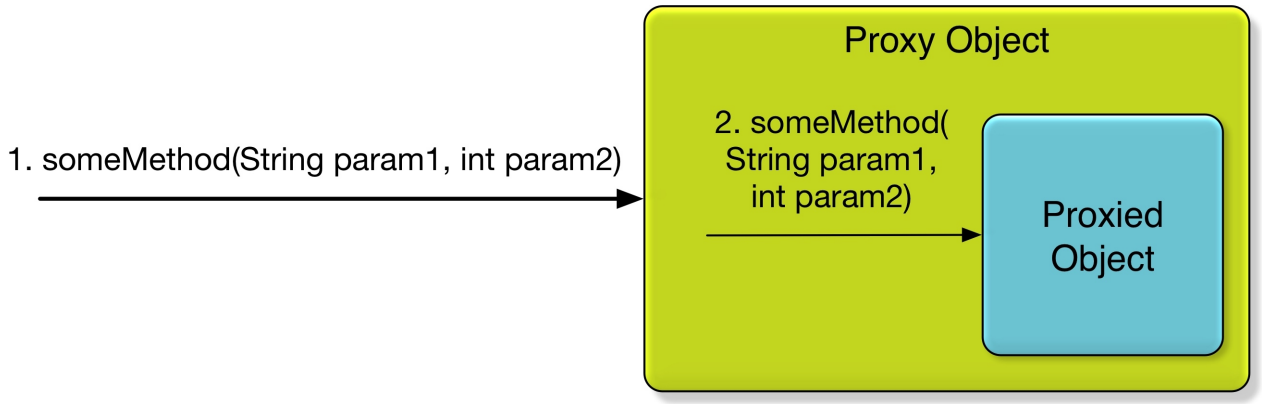
- Spring 5 API Documentation: @Qualifier
- Spring 5 Reference: Fine-tuning annotation-based autowiring with qualifiers
- Spring 5 Reference: Providing qualifier metadata with annotations

What is a proxy object and what are the two different types of proxies Spring can create?

For those familiar with the decorator design pattern, a proxy object is a decorator.

For those not familiar with the decorator design pattern or wanting a more detailed explanation:
A proxy object is an object that have the same methods, at least the public methods, as the object it proxies. The purpose of this is to make the proxy indistinguishable from the object it proxies. The proxy object contains a reference to the object it proxies. When a reference to the original, proxied, object is requested, a reference to the proxy object is supplied. When another object wants to invoke a method on the original object, it will invoke the same method on the proxy object. The proxy object may perform some processing before, optionally, invoking the (same) method on the original object.

In Spring, if a bean has been proxied, the proxy object will be supplied whenever the bean is requested.



Method invocation to proxied Java object where the same method is invoked on the proxied object.

Types of Spring Proxy Objects

The Spring framework is able to create two types of proxy objects:

- JDK Dynamic Proxy
  Creates a proxy object that implements all the interfaces implemented by the object to be proxied.
- CGLIB Proxy
  Creates a subclass of the class of the object to be proxied.

The default type of proxy used by the Spring framework is the JDK dynamic proxy.

What are the limitations of these proxies (per type)?

Each of the proxy types have certain limitations.

Limitations of JDK Dynamic Proxies

Limitations of JDK dynamic proxies are:

- Requires the proxied object to implement at least one interface.
- Only methods found in the implemented interface(s) will be available in the proxy object.
- Proxy objects must be referenced using an interface type and cannot be referenced using a type of a superclass of the proxied object type.
  This unless of course the superclass implements interface(s) in question.
  Requires care as far as types returned from @Bean methods and dependency-injected types are concerned.
- Does not support self-invocations.
  Self-invocation is where one method of the object invokes another method on the same object.

Limitations of CGLIB Proxies

Limitations of CGLIB proxies are:

- Requires the class of the proxied object to be non-final.
  Subclasses cannot be created from final classes.
- Requires methods in the proxied object to be non-final.
  Final methods cannot be overridden.
- Does not support self-invocations.
  Self-invocation is where one method of the object invokes another method on the same object.
- Requires a third-party library.
  Not built into the Java language and thus require a library. The CGLIB library has been included into Spring, so when using the Spring framework, no additional library is required.

What is the power of a proxy object and where are the disadvantages?

Some powers of proxy objects are:

- Can add behavior to existing beans.
  Examples of such behavior: Transaction management, logging, security.
- Makes it possible to separate concerns such as logging, security etc from business logic.

Some disadvantages of proxy objects are:

- It may not be obvious where a method invocation is handled when proxy objects are used.
  A proxy object may chose not to invoke the proxied object.
- If multiple layers of proxy objects are used, developers may need to take into account the order in which the proxies are applied.

- Can only throw checked exceptions as declared on the original method.
  Any recoverable errors in proxy objects that are not covered by checked exceptions declared on the original method may need to result in unchecked exceptions.
- Proxy object may incur overhead.
  Proxies that access external resources, use remote communication or perform lengthy processing may cause method invocations to become slower.
- May cause object identity issues.
  Proxy object and proxied object are two different objects and if using the equals operator (==) to compare objects, the result will be erroneous.

References:

- Spring 5 Reference: Proxying mechanisms
- Spring 5 Reference: AOP Concepts
- Spring 5 Reference: Declaring a Pointcut
- Java 8 API Documentation: Proxy
- Wikipedia: Decorator Design Pattern
- CGLIB
- Spring 5 Reference: AOP Proxies

What does the @Bean annotation do?

The @Bean annotation tells the Spring container that the method annotated with the @Bean annotation will instantiate, configure and initialize an object that is to be managed by the Spring container. In addition, there are the following optional configuration that can be made in the @Bean annotation:

- Configure autowiring of dependencies; whether by name or type.
- Configure a method to be called during bean initialization (initMethod).
  As before, this method will be called after all the properties have been set on the bean but before the bean is taken in use.
- Configure a method to be called on the bean before it is discarded (destroyMethod).
- Specify the name and aliases of the bean.
  An alias of a bean is an alternative bean-name that can be used to reference the bean.

The default bean name is the name of the method annotated with the @Bean annotation and it will be used if there are no other name specified for the bean.

References:

- Spring 5 API Documentation: @Bean
- Spring 5 Reference: Using the @Bean annotation
- Spring 5 Reference: Basic concepts: @Bean and @Configuration

What is the default bean id if you only use @Bean? How can you override this?

As in the previous section, the default bean name, also called bean id, is the name of the @Bean annotated method. This default id can be overridden using the name, or its alias value, attribute of the @Bean annotation.

References:

- Spring 5 API Documentation: @Bean

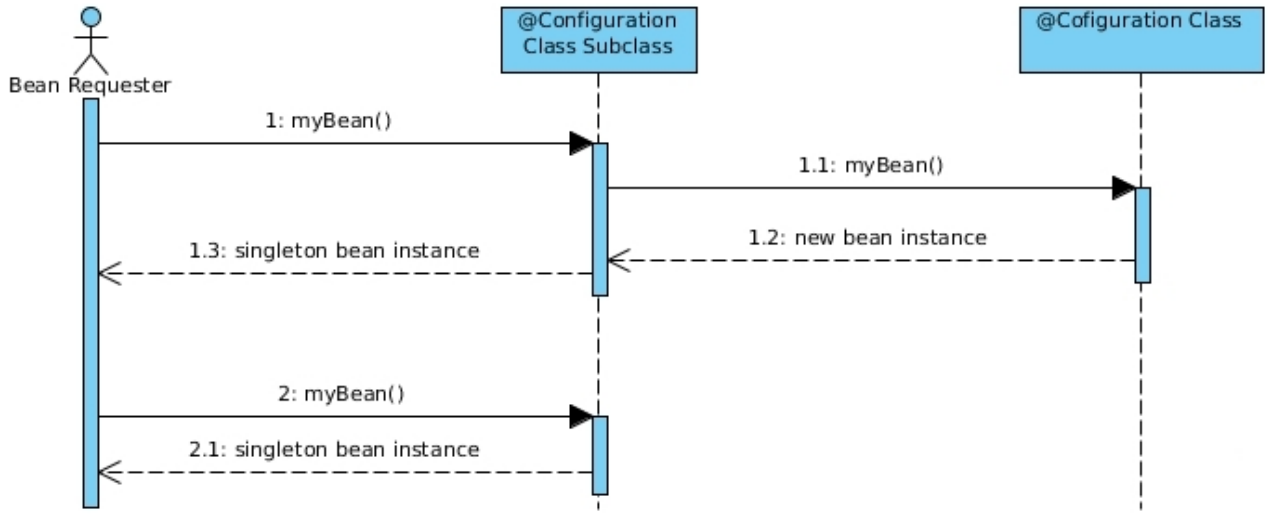
Why are you not allowed to annotate a final class with @Configuration?

The Spring container will create a subclass of each class annotated with @Configuration when creating an application context using CGLIB. Final classes cannot be subclassed, thus classes annotated with @Configuration cannot be declared as final.
The reason for the Spring container subclassing @Configuration classes is to control bean creation – for singleton beans, subsequent requests to the method creating the bean should return the same bean instance as created at the first invocation of the @Bean annotated method.

How do @Configuration annotated classes support singleton beans?

Singleton beans are supported by the Spring container by subclassing classes annotated with @Configuration and overriding the @Bean annotated methods in the class. Invocations to the @Bean annotated methods are intercepted and, if a bean is a singleton bean and no instance of the singleton bean exists, the call is allowed to continue to the @Bean annotated method, in order to create an instance of the bean. If an instance of the singleton bean already exists, the existing instance is returned (and the call is not allowed to continue to the @Bean annotated method).

The diagram below shows two subsequent requests for a singleton bean, handled as described above.



Two subsequent requests for a singleton bean to a @Configuration annotated class.

Why can’t @Bean methods be final either?

As earlier the Spring container subclass classes @Configuration classes and overrides the methods annotated with the @Bean annotation, in order to intercept requests for the beans. If the bean is a singleton bean, subsequent requests for the bean will not yield new instances, but the existing instance of the bean.

References:

- Spring 5 API Documentation: @Configuration
- Spring 5 Reference: Basic concepts: @Bean and @Configuration
- Spring 5 Reference: Using the @Configuration annotation

How do you configure profiles? What are possible use cases where they might be useful?

Bean definition profiles is a mechanism that allows for registering different beans depending on different conditions. Some examples of such conditions are:

- Testing and development
  Certain beans are only to be created when running tests. When developing, an in-memory database is to be used, but when deploying a regular database is to be used.
- Performance monitoring.
- Application customization for different markets, customers etc.

Configuring Profiles for Beans

One or more beans can be configured to be registered when one or more profiles are active using the @Profile annotation. The @Profile annotation can specify one or more profiles to which the bean(s) belong. Example:

```
@Profile({"dev", "qa"})
@Configuration
public class MyConfigurationClass {
…
```

The beans in the above configuration class will be registered if the “dev” or “qa” profile is active.

Profile names in the @Profile annotation can be prefixed with !, indicating that the bean(s) are to be registered when the the profile with specified name is not active. Example:

```
@Profile("!prod")
@Configuration
public class AnotherConfigurationClass {
…
```

The beans in the above configuration class will be registered if any profile except the “prod” profile is not active.

The @Profile annotation can be applied at the following locations:

- At class level in @Configuration classes.
  Beans in the configuration class and beans in configuration(s) imported with @Import annotation(s) will only be created and registered if the conditions in the @Profile annotation are met.
- At class level in classes annotated with @Component or annotated with any other annotation that in turn is annotated with @Component.
  The component will only be created and registered if the conditions in the @Profile annotation are met.
- On methods annotated with the @Bean annotation.
  Applied to a single method annotated with the @Bean annotations. The bean will only be created and registered if the conditions in the @Profile annotation are met. If overloading a

@Bean annotated method annotated with @Profile, the configuration of the @Profile annotation on the first overloaded method will be used if @Profile annotations are not consistent on the @Bean methods.

- Type level in custom annotations.
  Acts as a meta-annotation when creating custom annotations.

Beans that do not belong to any profile will always be created and registered regardless of which profile(s) are active.

Activating Profile(s)

One or more profiles can be activated using one of the following options:

- Programmatic registration of active profiles when the Spring application context is created.
- Using the spring.profiles.active property
- In tests, the @ActiveProfiles annotation may be applied at class level to the test class specifying which the profile(s) that are to be activated when the tests in the class are run.

The following example shows how to programmatically activate profiles when creating a Spring application context:

```
final AnnotationConfigApplicationContext theApplicationContext =
    new AnnotationConfigApplicationContext();
theApplicationContext.getEnvironment().setActiveProfiles("dev1", "dev2");
theApplicationContext.scan("se.ivankrizsan.spring");
theApplicationContext.refresh();
```

The following example shows how to launch an application in a JAR-file setting the active profiles using the spring.profiles.active property:

```
java -Dspring.profiles.active=dev1,dev2 -jar myApp.jar
```

There is a default profile named “default” that will be active if no other profile is activated.

References:

- Spring 5 Reference: Conditionally include @Configuration classes or @Bean methods
- Spring 5 Reference: Bean definition profiles
- Spring 5 Reference: Activating a profile
- Spring 5 API Documentation: @Profile
- Spring 5 API Documentation: @ActiveProfiles

Can you use @Bean together with @Profile?

Yes, see section above on how you configure profiles.

Can you use @Component together with @Profile?

Yes, see section above on how you configure profiles.

How many profiles can you have?

There does not seem to be any limitation concerning how many profiles that can be used in a Spring application. The Spring framework (in the class *ActiveProfilesUtils*) use an integer to iterate over an array of active profiles, which implies a maximum number of 2^32 – 1 profiles.

How do you inject scalar/literal values into Spring beans?

Scalar/literal values can be injected into Spring beans using the @Value annotation. Such values can originate from environment variables, property files, Spring beans etc.
This example shows how the value of the personservice.retry-count from a property file is injected into a field of a Spring bean. Note the ${} construct that surrounds the name of the property which value is to be injected.

```
@Component
public class MyBeanClass {
    @Value("${personservice.retry-count}")
    protected int personServiceRetryCount;
```

A default value can be specified if the property value to be injected is not available. Even another property value can be used as default value like in this example:

```
@Component
public class MyBeanClass {
    @Value("${personservice.retry-count:${services.default.retry-count}}")
    protected int personServiceRetryCount;
```

The next example shows how the value of a SpEL expression is injected into a field of a Spring bean. Note that in this case, the expression is surrounded by the characters #{}.

```
@Component
public class MyBeanClass {
    @Value("#{ T(java.lang.Math).random() * 50.0 }")
    protected double randomNumber;
```

Please see subsequent sections for more detail on SpEL, the Spring Expression Langauge.

The @Value annotation can be applied to:

- Fields
- Methods
  Typically setter methods
- Method parameters
  Including constructor parameters. Note that when annotating a parameter in a method other than a constructor, automatic dependency injection will not occur. If automatic injection of the value is desired, the @Value annotation should be moved to the method instead.
- Definition of annotations
  In order to create a custom annotation.

References:

- Spring 5 Reference: Expression support for defining bean definitions
- Spring 5 API Documentation: @Value
- Spring 5 Reference: Annotation-based configuration

What is @Value used for?

The @Value annotation can be used for:

- Setting (default) values of bean fields, method parameters and constructor parameters.
  Note that no value will be injected if an object method parameter is annotated with @Value and the method is invoked with a null value for the parameter in question – the method parameter will have the value null.
- Injecting property values into bean fields, method parameters and constructor parameters.
- Injecting environment variable values into bean fields, method parameters and constructor parameters.
- Evaluate expressions and inject the result into bean fields, method parameters and constructor parameters.
- Inject values from other Spring beans into bean fields, method parameters and constructor parameters.
  This is a special case of evaluating expressions, where an expression specifying a Spring bean name and the name of a field in the bean is evaluated and the value injected into the target.

Note that Spring configuration classes are also instantiated as Spring beans, so dependency injection of values and references work as in any other Spring bean.

Please also refer to the previous section for more information.

References:

- Spring 5 Reference: Expression support for defining bean definitions
- Spring 5 API Documentation: @Value
- Spring 5 Reference: Annotation-based configuration

What is Spring Expression Language (SpEL for short)?

Note: All the examples in this section has been tested using the standalone Spring Expression Language parser and expression classes. If used in a @Value annotation, these example-expression must be surrounded with #{<insert expression here>} in order to work as expected.

The Spring Expression Language is an expression language used in the different Spring products, not only in the Spring framework. In addition, SpEL can be used SpEL has support for:

- Literal expressions.
  Types of literal expressions supported: Strings, numeric values, booleans and null.
  Example string: 'Hello World'
- Properties, arrays, lists and maps.
  Example create a list of integers: {1, 2, 3, 4, 5}
  Example create a map: {1 : "one", 2 : "two", 3 : "three", 4 : "four"}
  Example retrieve third item in list referenced by variable theList: #theList[3]
  Example retrieve value from map in variable personsMap that has key “ivan”: #personsMap['ivan']
  Example retrieve OS name system property: @systemProperties['os.name']
- Method invocation.
  Example invoke a method on a Java object stored in variable javaObject: #javaObject.firstAndLastName()
- Operators.
  Relational operators, logical operators, mathematical operators, assignment, type-operator.
  Creating Java objects using new operator.
  Ternary operator: <condition> ? <true-expression> : <false-expression>
  The Elvis operator: <variable-to-test-for-null> ?: <value-to-assign-if-variable-is-null>
  Safe navigation operator: <object-reference-that-may-be-null>?.<field-in-object>
  Regular expression “matches” operator: '168' matches '\\d+'
- Variables.
  Variables are set on the evaluation context and accessed in SpEL expressions using the # prefix.
  Example access the list in the numbersList variable: #numbersList
- User defined functions.
  Implemented as static methods.
- Referencing Spring beans in a bean factory (application context).
  Bean names are prefixed with @ in SpEL expressions.
  Example access the field injectedValue on the bean mySuperComponent: @mySuperComponent.injectedValue

- Collection selection expressions.
  Creates a new collection by selecting a subset of elements from a collection.
  Syntax: .?[<selection-expression>]
  Example collect entries in map where keys are even: #theMap.?[key % 2 == 0]
- Collection projection.
  Creates a new collection by applying an expression to each element in a collection.
  Syntax: .![<expression>]
  Example create a list of string representation of entries in a map: #theMap.!['Key: ' + key + ', Value: ' + value]
- Expression templating.
  An expression template is a text string that contains one or more SpEL expressions.
  Please refer to the complete example below.

Complete Standalone Expression Templating Examples

The following is a complete standalone test showing how to evaluate a SpEL expression in which templating is used. Note that a bean factory resolver, created with a reference to the current application context, and a template parser context is required in order for the expression used in the example to work.

```
    @Autowired
    protected ApplicationContext mApplicationContext;
    @Test
    public void expressionTemplatingTest() {
        /*
         * The bean factory resolver is needed to resolve the systemProperties
         * bean in the example expression.
         */
        final BeanFactoryResolver theBeanFactoryResolver =
            new BeanFactoryResolver(mApplicationContext);
        /* A template parser context is required with expression templating. */
        final TemplateParserContext theParserContext = new TemplateParserContext();
        final SpelExpressionParser theExpressionParser = new SpelExpressionParser();
        final Expression theExpression = theExpressionParser.parseExpression(
            "This computer uses the #{@systemProperties['os.name']} operating system.",
            theParserContext);

        final StandardEvaluationContext theEvaluationContext =
            new StandardEvaluationContext();
        theEvaluationContext.setBeanResolver(theBeanFactoryResolver);

        final Object theExpressionValue = theExpression.getValue(theEvaluationContext);
        System.out.println("Value: " + theExpressionValue);
```

```
        System.out.println("Value class: " + theExpressionValue.getClass().getName());
    }
```

If I run the above on my computer, which uses Ubuntu Linux, the output from the above print statements will be:

```
Value: This computer uses the Linux operating system.
Value class: java.lang.String
```

Compiled SpEL Expressions

SpEL expressions are evaluated dynamically as per default but can be compiled to yield improved performance. There are some limitations as to the expressions that can be compiled and the following types of expressions cannot be compiled:

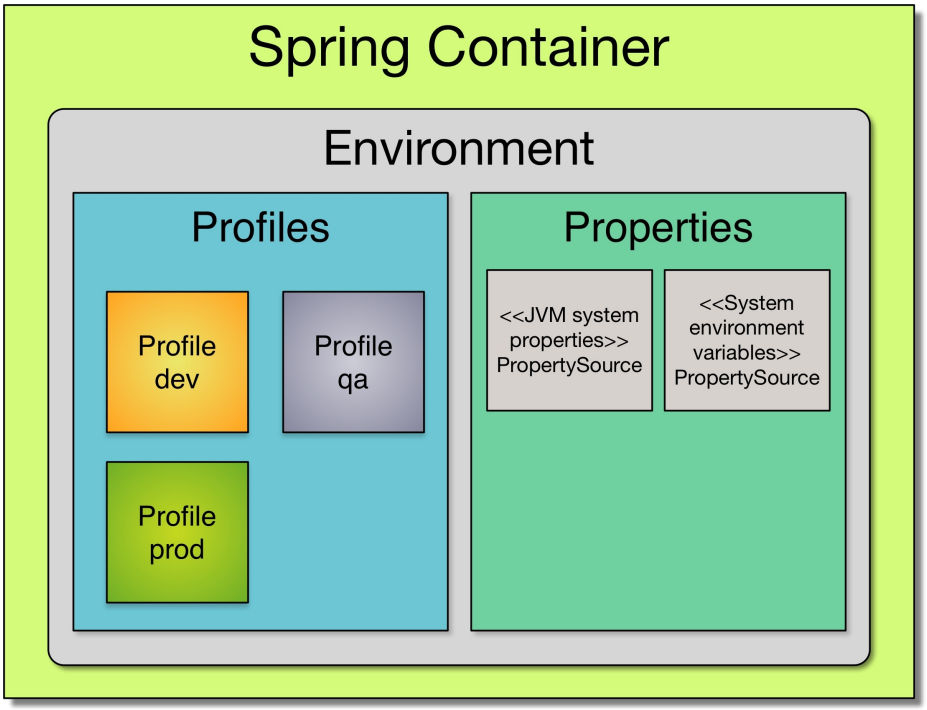
- Expressions involving assignment.
- Expressions relying on the conversion service.
- Expressions using custom resolvers or accessors.
  Custom resolvers and accessors, if used, are to be registered on an instance of the *StandardEvaluationContext* class.
- Expressions using selection or projection.

References:

- Spring 5 Reference: Spring Expression Language (SpEL)
- The Apache Groovy Programming Language: Elvis operator
- The Apache Groovy Programming Language: Safe navigation operator

What is the *Environment* abstraction in Spring?

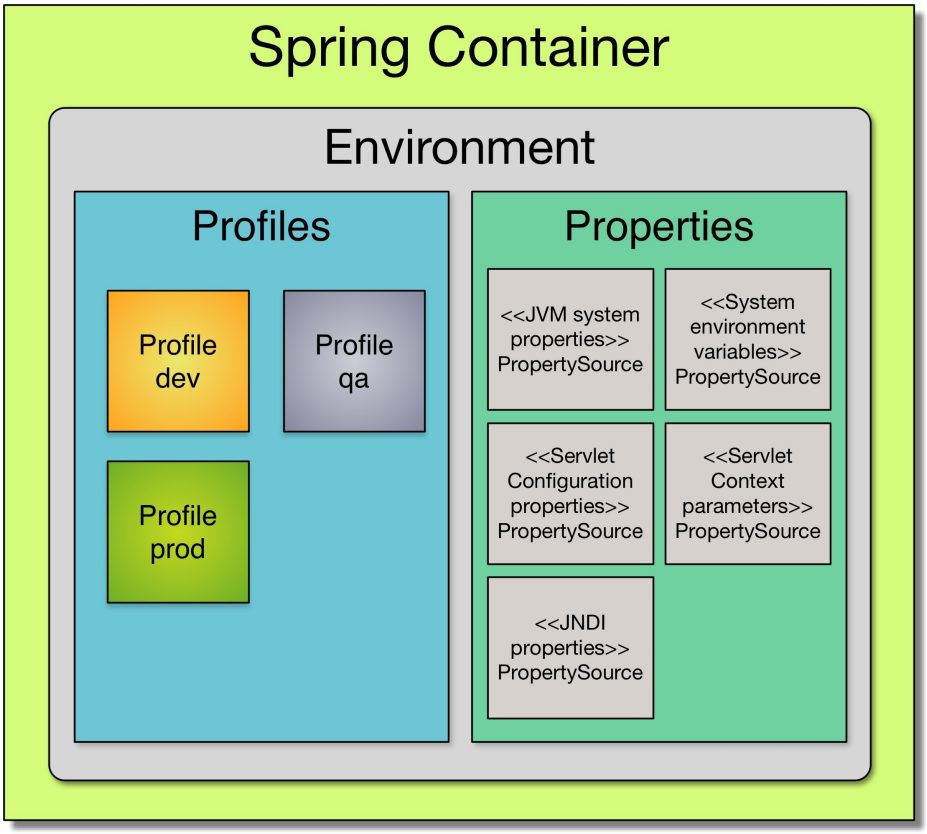
The Environment is a part of the application container. The Environment contains profiles and properties, two important parts of the application environment.
The following figure shows the Environment in a non-web application.



Environment containing profiles and properties in a non-web Spring application.

The Environment in a Spring application, both web- and non-web-applications, contains a number of profile, each containing a number of beans. Please see How do you configure profiles and related sections above for more information on bean profiles.
In addition the Environment contains a number of property sources. In a non-web application environment, there are two default property sources. The first contains the JVM system properties and the second contains the system environment variables.

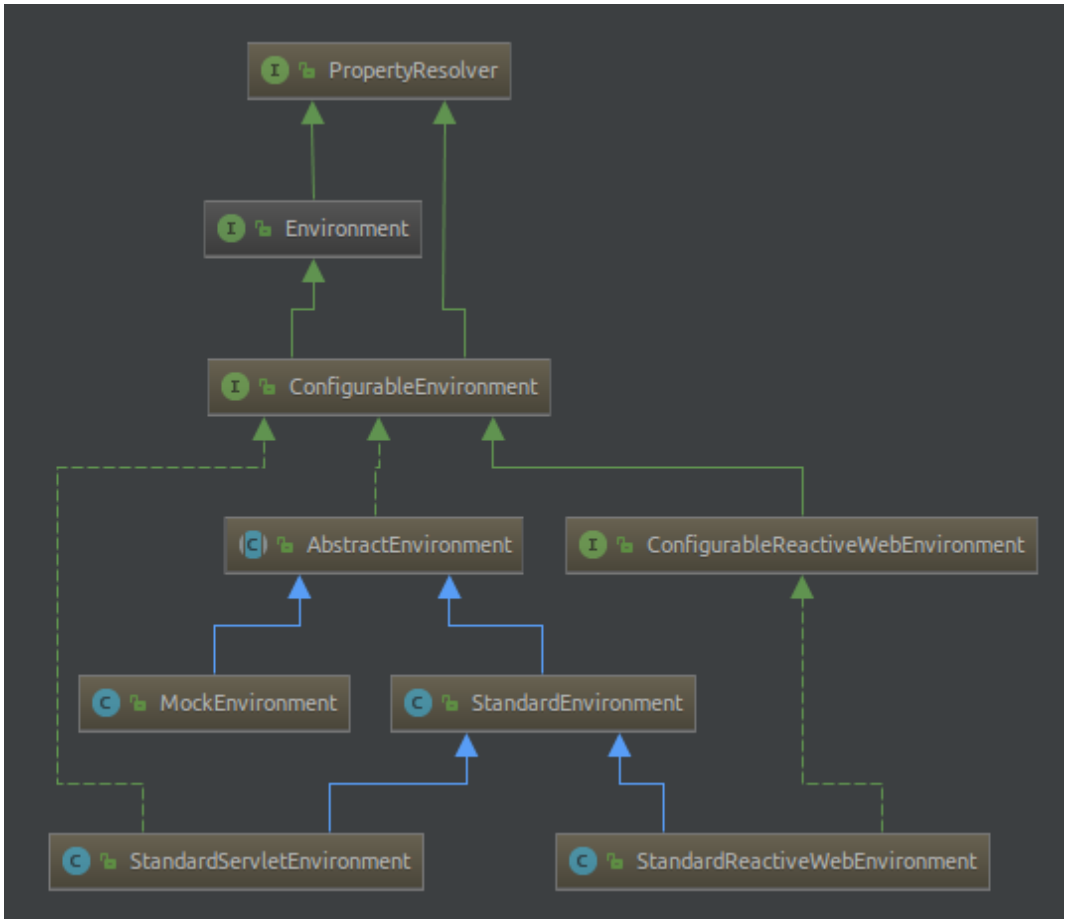
The environment of a servlet-based web application contains three additional default property sources as can be seen in the following figure.



Environment containing profiles and properties in a servlet based Spring web application.

Environment Class Hierarchy

The Environment class hierarchy looks like in the following class diagram.



Spring *Environment* class diagram.

The *StandardEnvironment* class is the basic concrete environment implementation for non-web applications. Both the *StandardServletEnvironment* and *StandardReactiveWebEnvironment* classes are subclasses of *StandardEnvironment*. These classes implement environments for regular servlet-based web applications and reactive web applications respectively.

Relation Between Application Context and Environment

The Spring *ApplicationContext* interface extends the *EnvironmentCapable* interface, which contain one single method namely the *getEnvironment* method, which returns an object implementing the *Environment* interface. Thus a Spring application context has a relation to one single *Environment* object.

References:

- Spring 5 Reference: Environment abstraction
- Spring 5 API Documentation: ApplictionContext

- Spring 5 API Documentation: EnvironmentCapable
- Spring 5 API Documentation: Environment
- Spring 5 API Documentation: StandardEnvironment
- Spring 5 API Documentation: StandardServletEnvironment

Where can properties in the environment come from – there are many sources for properties – check the documentation if not sure. Spring Boot adds even more.

The following table shows different property sources and the environment, if applicable, in which the property source first appears.

Property Source	Originating Environment
JVM system properties	StandardEnvironment
System environment variables	StandardEnvironment
Servlet configuration properties (ServletConfig)	StandardServletEnvironment
Servlet context parameters (ServletContext)	StandardServletEnvironment
JNDI properties	StandardServletEnvironment
Command line properties	n/a
Application configuration (properties file)	n/a
Server ports	n/a
Management server	n/a

References:

- Spring 5 API Documentation: StandardEnvironment
- Spring 5 API Documentation: StandardServletEnvironment
- Spring Boot 2 Reference: Accessing Command Line Properties
- Spring Boot 2 Reference: Accessing Application Arguments

What can you reference using SpEL?

The following entities can be referenced from Spring Expression Language (SpEL) expressions.

- Static methods and static properties/fields.
  Example invoke a static method on a class:
  T(se.ivankrizsan.spring.MyBeanClass).myStaticMethod()
  Example access contents of static field (class variable) on a class:
  T(se.ivankrizsan.spring.MyBeanClass).myClassVariable
- Properties and methods in Spring beans.
  A Spring bean is references using its name prefixed with @ in SpEL. Chains of property references can be accessed using the period character.
  Example accessing property on Spring bean: @mySuperComponent.injectedValue
  Example invoking method on Spring bean: @mySuperComponent.toString()
- Properties and methods in Java objects with references stored in SpEL variables.
  References and values stored in variables are referenced using the variable name prefixed with # in SpEL.
  Example accessing property on Java object: #javaObject.firstName
  Example invoking method on Java object: #javaObject.firstAndLastName()
- (JVM) System properties.
  Available through the systemProperties reference, which is available by default.
  Example retrieving OS name property: @systemProperties['os.name']
- System environment properties.
  Available through the systemEnvironment reference, which is available by default.
  Example KOTLIN_HOME environment variable:
  @systemEnvironment['KOTLIN_HOME']
- Spring application environment.
  Available through the environment reference, also available by default.
  Example retrieve name of first default profile: @environment['defaultProfiles'][0]

Additional references can be added depending on context and what parts of the Spring eco-system used by the application.

References:

- Spring 5 Reference: Spring Expression Language (SpEL)
- Spring 5 API Documentation: ConfigurableApplicationContext

What is the difference between $ and # in @Value expressions?

Expressions in @Value annotations are of two types:

- Expressions starting with $.
  Such expressions reference a property name in the application's environment. These expressions are evaluated by the *PropertySourcesPlaceholderConfigurer* Spring bean prior to bean creation and can only be used in @Value annnotations.
- Expressions starting with #.
  Spring Expression Language expressions parsed by a SpEL expression parser and evaluated by a SpEL expression instance.

References:

- Spring 5 Reference: Expression support for defining bean definitions
- Spring 5 API Documentation: PropertySourcesPlaceholderConfigurer
- Spring 5 API Documentation: SpelExpressionParser
- Spring 5 API Documentation: SpelExpression