Core Spring 5 Certification in Detail

Ivan Krizsan

Version: 2020-02-02

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?

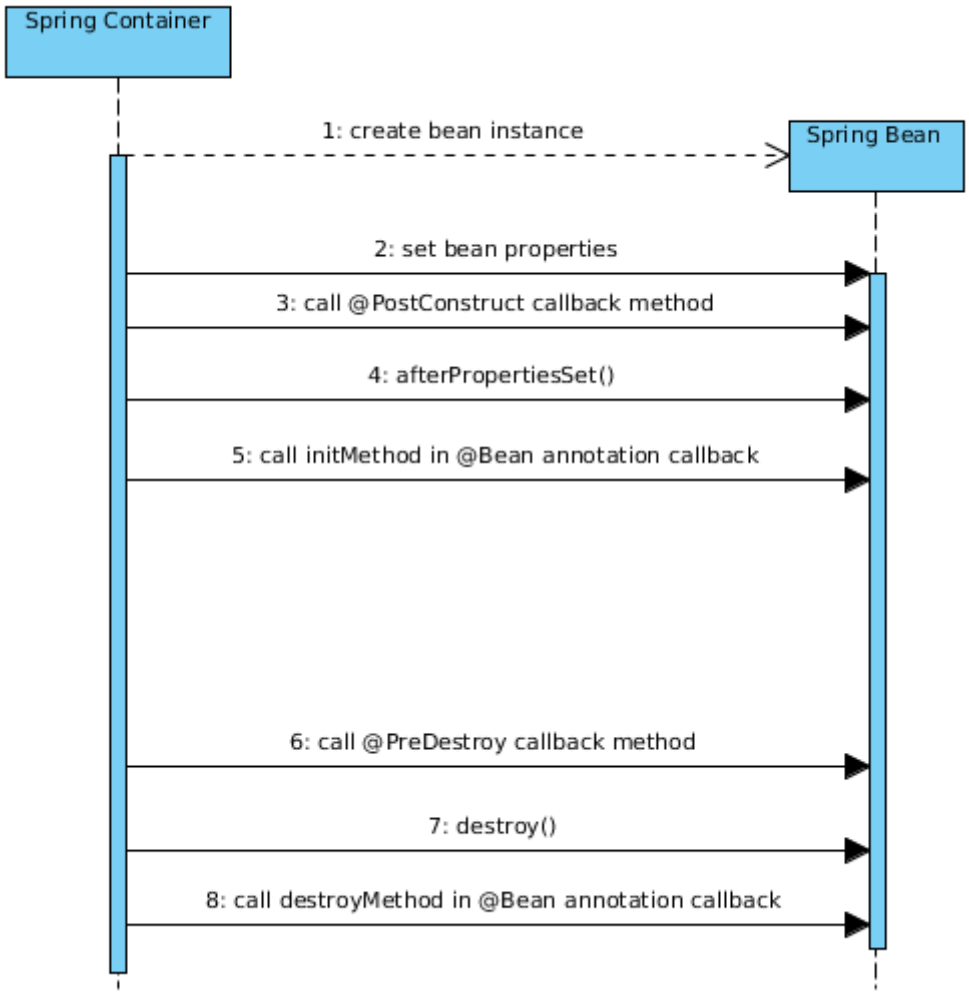
An initialization method is a method in a Spring bean that will be invoked by the Spring application container after all properties on the bean have been populated but before the bean is taken into use. An initialization method allow the Spring bean to perform any initialization that depend on the properties 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

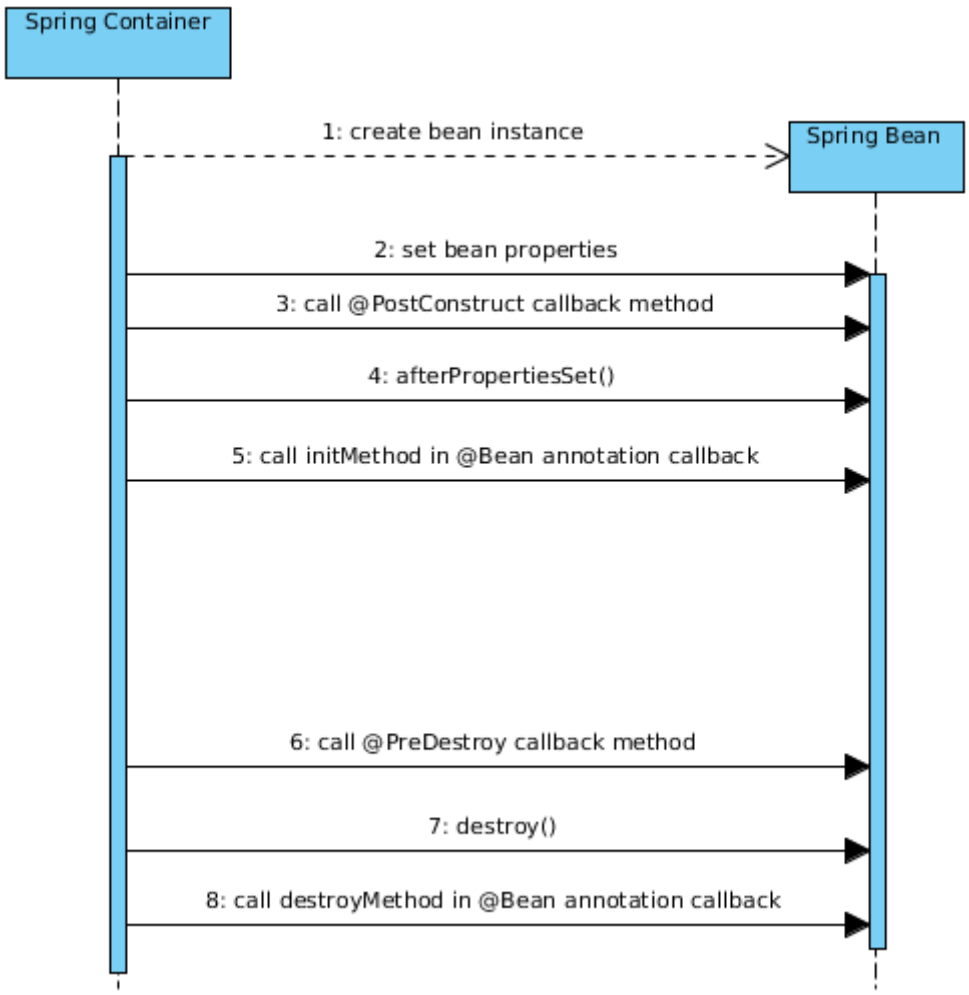
What is a destroy method, how is it declared and when is it called?

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 these different locations:

- At injection points
- At bean definitions
- At stereotype annotations
  Classes annotated with stereotype annotation is a type of bean definition.
- 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 Stereotype Annotations

Similar to qualifiers at bean definitions, the @Qualifier annotation can also be used at the same place, that is class level, to accompany stereotype annotations like @Component, @Repository, @Service etc. This will have the same effect as annotating a bean definition with the @Qualifier annotation and the same qualifier can be used at an injection point to inject the bean created from the annotated component, repository, service etc.

@Qualifier at Annotation Definitions

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

References:

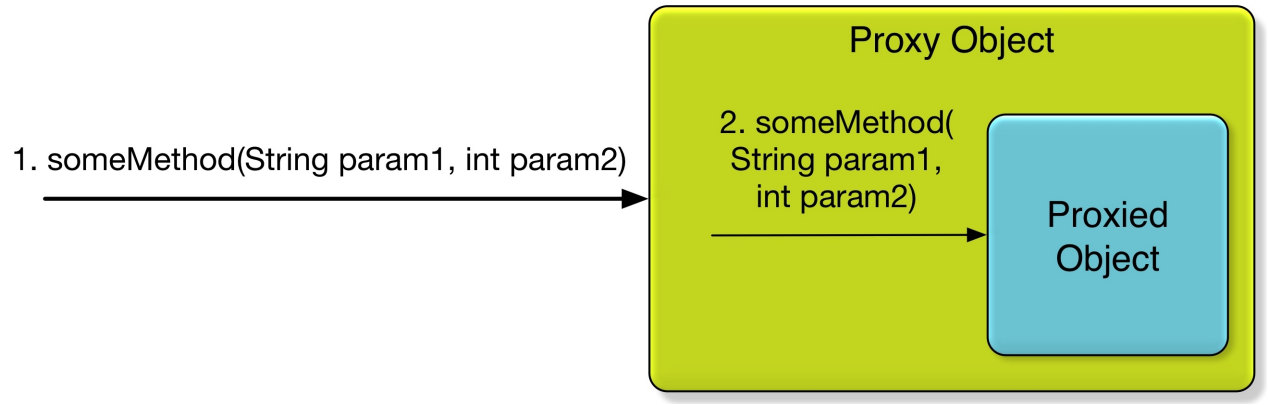
- Spring 5 API Documentation: @Qualifier
- Spring 5 Reference: Fine-tuning annotation-based autowiring with qualifiers
- Spring 5 Reference: Providing qualifier metadata with annotations

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

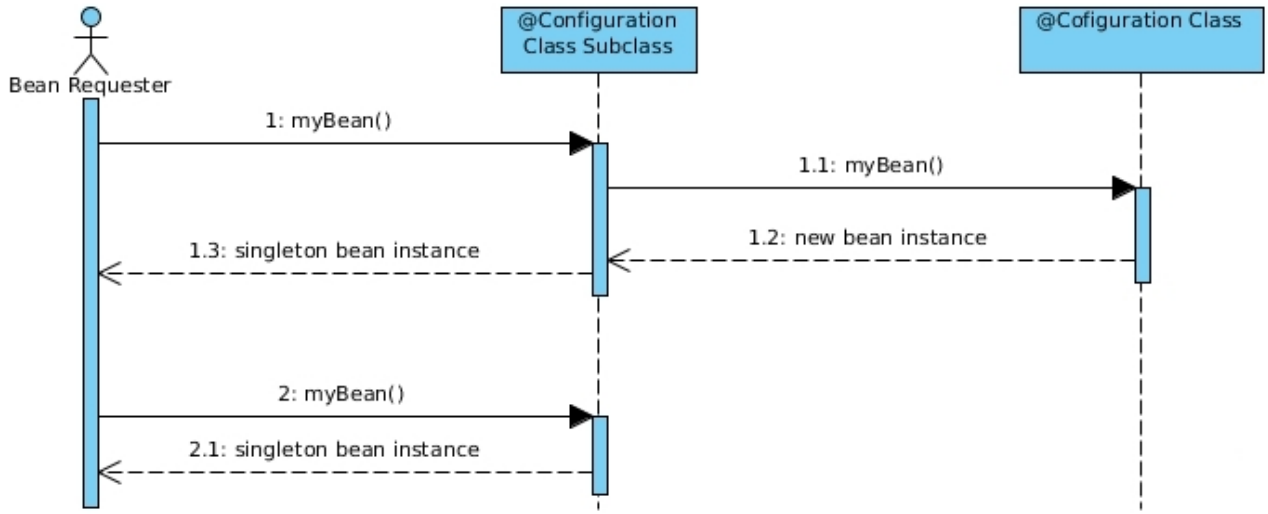
Why are you not allowed to annotate a final class with @Configuration?

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 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:

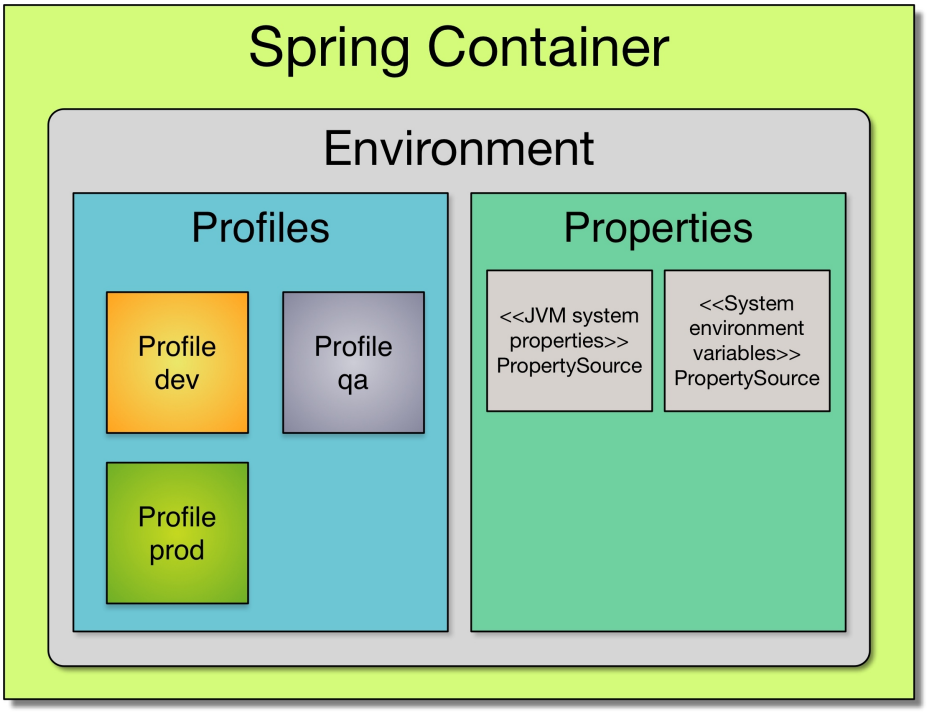
- Expressions involving assignment.
- Expressions relying on the conversion service.
- Expressions using custom resolvers or accessors.
  Custom resolvers and accessors, if used, are to be registered on an instance of the *StandardEvaluationContext* class.
- Expressions using selection or projection.

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?

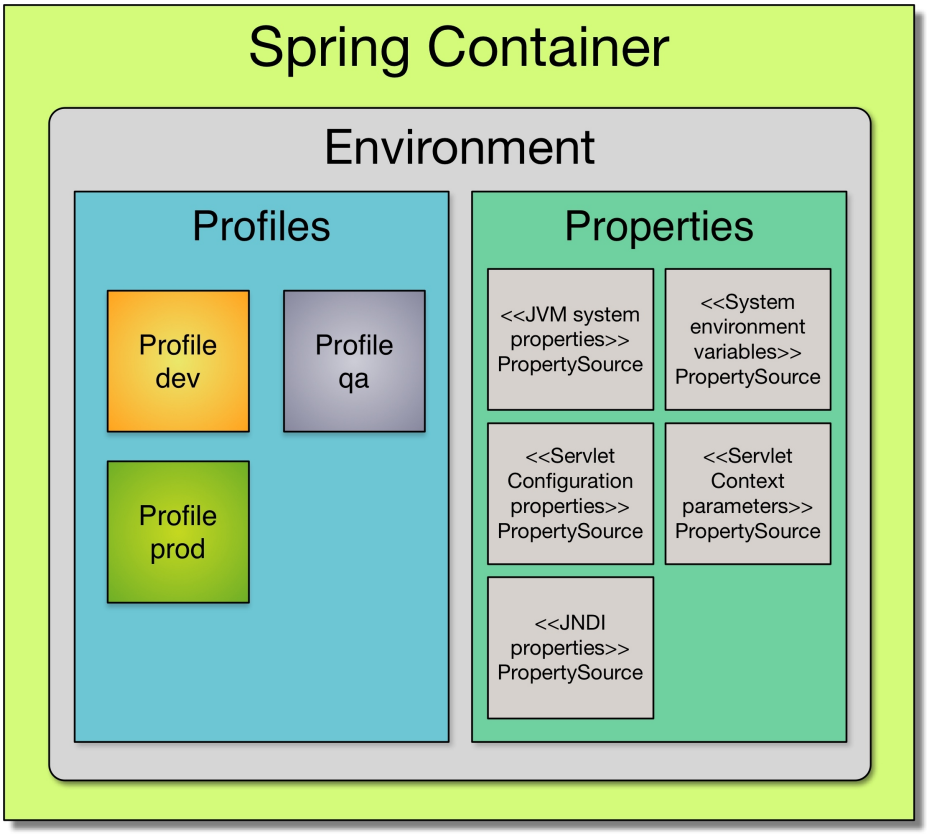
The Environment is a part of the application container. The Environment contains profiles and properties, two important parts of the application environment.
The following figure shows the Environment in a non-web application.



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.

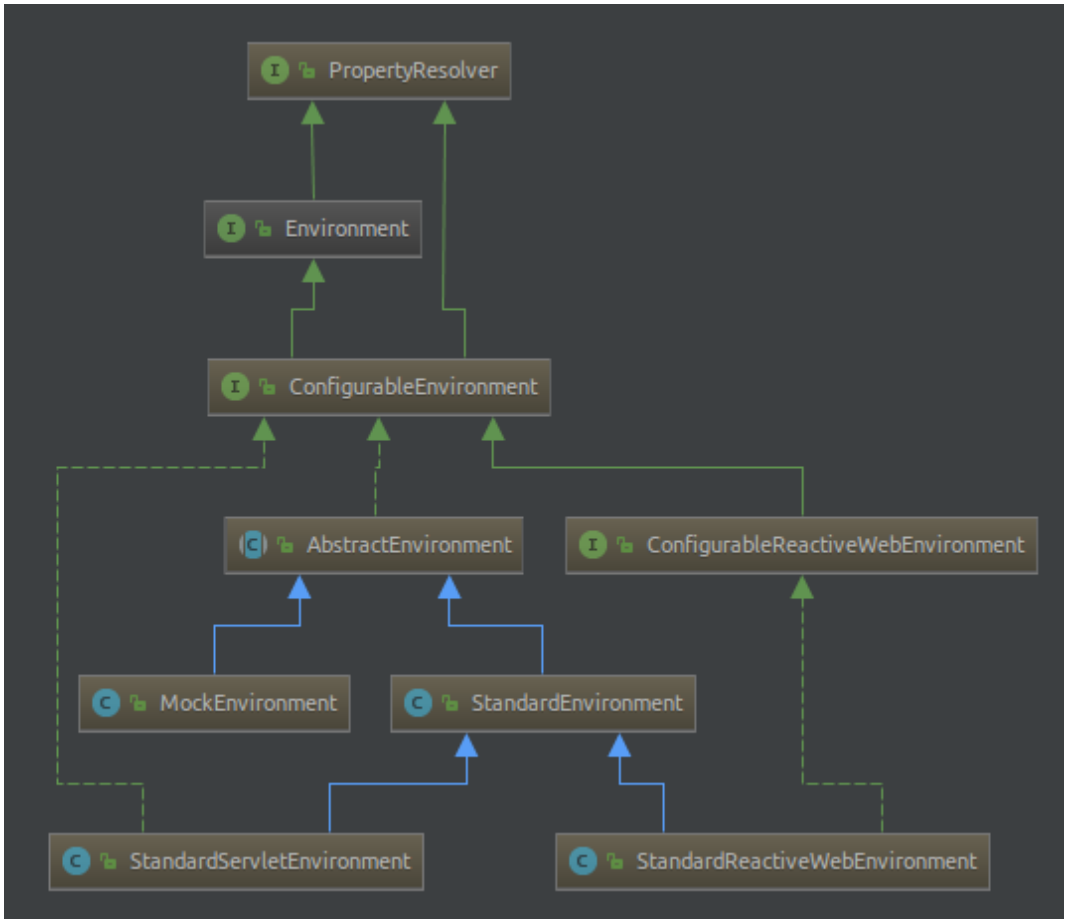
The environment of a servlet-based web application contains three additional default property sources as can be seen in the following figure.



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

Chapter 2

Aspect Oriented Programming

What is the concept of AOP? Which problem does it solve? What is a cross cutting concern?

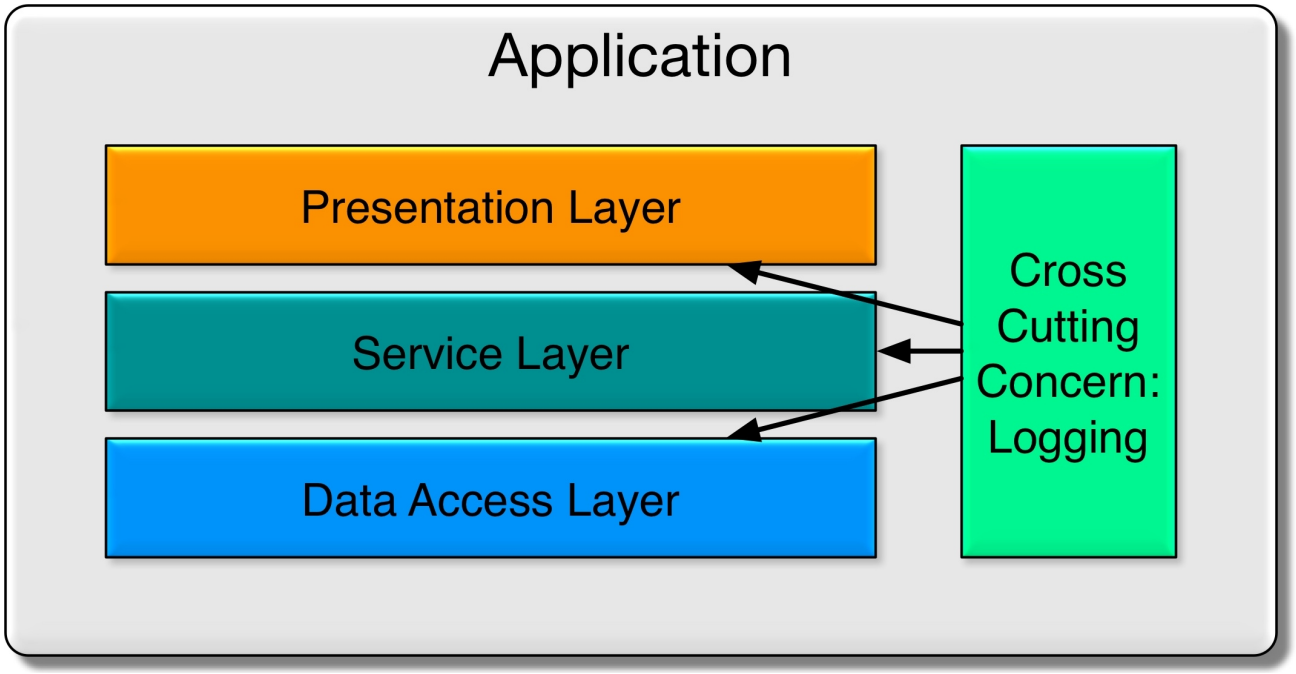
What is the concept of AOP?

AOP is short for Aspect Oriented Programming. AOP is a programming paradigm that aims at separating a group of concerns in a software application, so called cross cutting concerns, from other code. This is accomplished by specifying the following:

- Location(s) in the code where code of the cross cutting concern is to be inserted.
- Code of the cross cutting concern.

Both the above are specified separated from the code at which the cross cutting concern is applied.

The figure below shows a cross cutting concern, logging in this example, separated from the layered architecture of the application. The cross cutting concern specifies the locations in the application code at which the custom behavior is to be inserted and also contains the implementation of the custom behavior.



Application using a layered architecture with a cross cutting concern, in this case logging, adding behavior to classes in all three layers.

What is a cross cutting concern?

Cross cutting concern is functionality that is used in multiple locations throughout an application. Such functionality is applied/used in areas of an application that normally are not related to each other. Commonly such concerns are not related to business logic, but there are exceptions.
Cross cutting concerns do not usually fit well in the model of object-oriented or procedural programming. An example of a cross cutting concern is logging.
Logging is applied throughout the application in areas that are not related. Examples of such areas are:

- Data layer and presentation layer.
- Application code and third-party library code.

Name three typical cross cutting concerns.

Examples of concerns that are usually cross cutting concerns are:

- Logging
- Caching
- Security
- Data validation
- Transaction management

- Error handling
- Monitoring
  An example is monitoring of the processing time of (certain) methods in an application.
- Custom business rules

Which problems does AOP solve?

AOP as a concept can solve the following problems:

- Avoid code duplication.
  Instead of duplicating for instance transaction handling code throughout the application, it can be implemented in one single place using AOP.
- Avoid mixing of, for instance, business logic and cross cutting concerns.
  AOP enables separation of concerns by allowing a cross cutting concerns to be implemented in a module of its own.

The following are some examples of situations when using AOP can be useful:

- Implementing cross cutting concerns in an application.
- Implementing additional functionality in or customizing (legacy/third-party) code.
  At times legacy or third-party code does not allow for modification without significant effort and/or having to modify unrelated code (when methods affected by modification are private).
  It may also be desirable to separate customizations and other modifications from the code customized and/or modified, for instance with product development, as to reduce the effort needed when the core product changes.
- Tracing code in order to, for instance, correct bugs.

What two problems arise if you don't solve a cross cutting concern via AOP?

If cross cutting concerns were to be implemented in an application without using AOP the following are very likely to happen:

- Code duplication
  Same, or similar, code duplicated in several locations.
- Mixing of concerns
  Code of the cross cutting concern, for instance transaction handling, would become mixed with service layer code etc. Makes the code more difficult to read and maintain.

References:

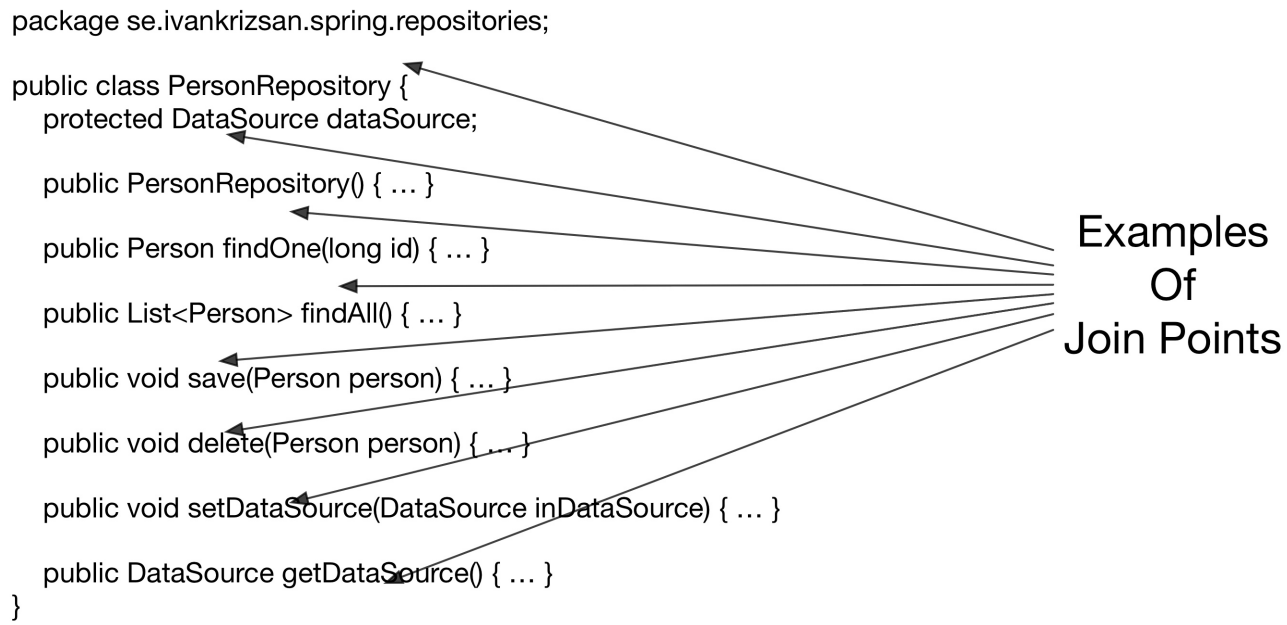
- Wikipedia: Aspect Oriented Programming

- Wikipedia: Cross-Cutting Concern

What is a pointcut, a join point, an advice, an aspect, weaving?

Join Point

A join point is a point in the execution of a program at which additional behavior can be inserted using AOP. The following figure shows examples of join points in a Java class.



Examples of join points in a Java class.

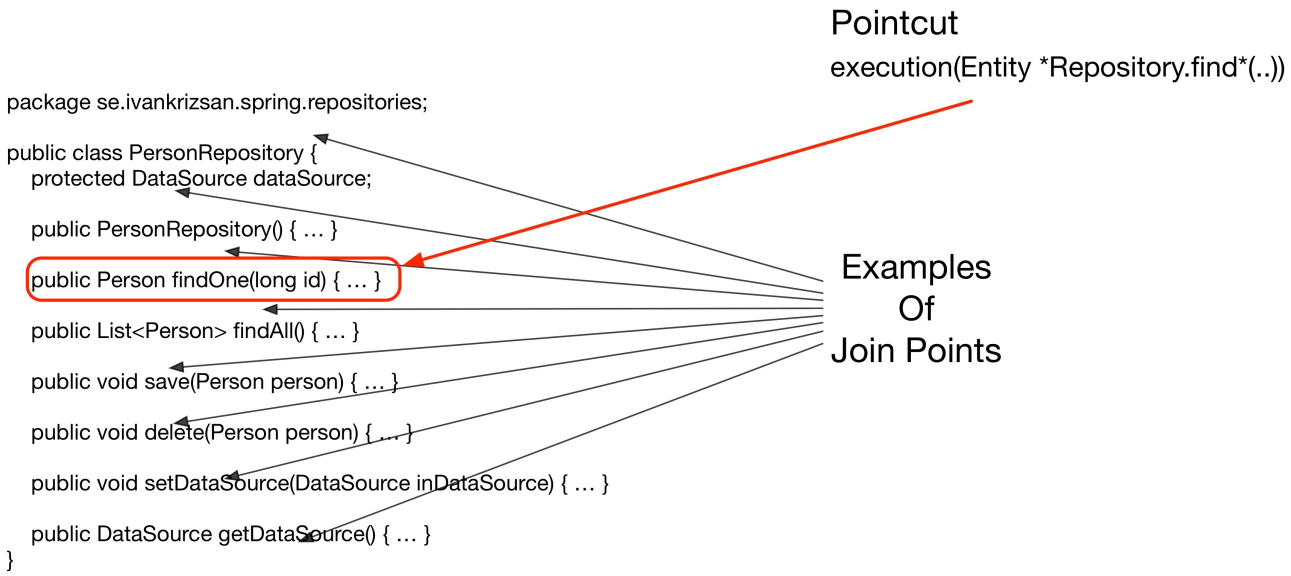
Some join point types are (the list is not exhaustive):

- Method invocation
- Method execution
- Object creation
- Constructor execution
- Field references

Spring AOP only supports public method invocation join points. Compare to AspectJ which supports all of the above listed join point types and more.

Pointcut

A pointcut selects one or more join points out of the set of all join points in an application.



Pointcut selecting one single join point in the PersonRepository class.
Note that the findAll method is not selected since it returns a list, not an entity.
The pointcut may still select additional join points in other classes.

Example:

```
execution(Entity *Repository.find*(..))
```

The above pointcut selects all the join points that is an execution of a method which name starts with “find” with zero or more parameters on an object of a class which name ends with “Repository” that returns an object of the type *Entity*.

Pointcuts can be combined using the logical operators && (and), || (or) and ! (not):

```
execution(void *Repository.save(..)) &&
within(se.ivankrizsan.spring..*)
```

The above pointcut combines two pointcuts and selects all join points that is execution of the *save* method on an object of a class which name ends in “Repository” and that is located within the package se.ivankrizsan.spring, including sub-packages.

As seen, wildcards can be used in pointcut expressions:

- *
  Matches types (return and argument types), whole or partial names (package, class, method).
- ..
  Matches zero or more arguments or packages.

The following is an example of Spring AOP where multiple pointcuts are combined and used with an around advice:

```
/**
 * Spring AOP example: Logging aspect.
 *
 * @author Ivan Krizsan
 */
@Aspect
@Component
public class LoggingAspect {
    /**
     * Pointcut that selects join points being method executions in the se package
     * and sub-packages in classes which name ends with "Service" having arbitrary
     * number of parameters.
     */
    @Pointcut("execution(* se..*.*Service.*(..))")
    public void applicationServiceMethodPointcut() {}

    /**
     * Pointcut that selects join points being public method executions.
     */
    @Pointcut("execution(public * *(..))")
    public void publicMethodPointcut() {}

    /**
     * Pointcut that selects join points within the package se.ivankrizsan.spring
     * and sub-packages.
     */
    @Pointcut("within(se.ivankrizsan.spring..*)")
    public void inSpringPackagePointcut() {}

    /**
     * Pointcut that selects join points on the Spring bean with the
     * name "mySuperService".
     */
    @Pointcut("bean(mySuperService)")
    public void mySuperServiceSpringBeanPointcut() {}

    /**
     * Pointcut that combines all the above pointcuts to select join points
     * that match all the pointcuts.
     */
```

```
    @Pointcut("publicMethodPointcut() && inSpringPackagePointcut() &&
applicationServiceMethodPointcut() && mySuperServiceSpringBeanPointcut()")
    public void publicServiceMethodInSpringPackagePointcut() {}

    /**
     * Around advice that prints a string before and after execution of a
     * join point (typically a method).
     *
     * @param inProceedingJoinPoint Join point advised by around advice.
     * @return Result from the join point invocation.
     * @throws Throwable If exception thrown when executing join point.
     */
    @Around("publicServiceMethodInSpringPackagePointcut()")
    public Object loggingAdvice(
        final ProceedingJoinPoint inProceedingJoinPoint) throws Throwable {
        System.out.printf("* Before service method '%s' invocation.%n",
            inProceedingJoinPoint.getSignature().toShortString());

        /* Invoke the join point. */
        final Object theResult = inProceedingJoinPoint.proceed();

        System.out.printf("* After service method '%s' invocation.%n",
            inProceedingJoinPoint.getSignature().toShortString());

        return theResult;
    }
}
```

Note the definition of the different pointcuts and finally how a pointcut is defined that refers the previously defined pointcuts. The last pointcut is then used in an around advice to specify when the advice is to be executed.
Please refer to subsequent sections on advice and the *ProceedingJoinPoint* class.

Advice

Advice is the additional behavior, typically a cross cutting concern, that is to be executed at certain places (at join points) in a program. The example in the above Pointcut section shows an around advice that prints a message before and after the execution of a join point.

Please refer to subsequent sections that describes the different types of advice available and how advice is implemented in Spring AOP.

Aspect

An aspect brings together one or more pointcuts with one or more advice. Typically one aspect encapsulates one cross cutting concern, as to adhere to the single responsibility principle.

The logging aspect given as an example in the above section on pointcuts is a good example, although the pointcuts are unnecessarily complicated for the sake of showing different types of pointcuts.

Weaving

Weaving is the process by which aspects and (application) code is combined as to enable execution of cross cutting concerns at the join points specified by the pointcuts in the aspects.
Weaving can occur at different points in time:

- Compile time weaving
  Byte code of classes is modified at compilation time at selected join points to execute advice code. The AspectJ compiler uses compile time weaving.
- Load time weaving
  Byte code of classes is modified at class loading time when the application is run.
- Runtime weaving
  Proxy objects are created at runtime when the application is run. The proxy objects are used instead of the original objects and divert method invocations to advice code.
  Spring AOP uses runtime weaving exclusively.

References:

- Spring 5 Reference: AOP concepts
- The AspectJ Programming Guide
- Spring 5 Reference: @AspectJ support
- The AspectJ Programming Guide: Quick Reference: Pointcuts
- The AspectJ 5 Development Kit Developer’s Notebook

How does Spring solve (implement) a cross cutting concern?

Spring uses proxy objects to implement the method invocation interception part of AOP. Such proxy objects wrap the original Spring bean and intercepts method invocations as specified by the set of pointcuts defined by the cross cutting concern.

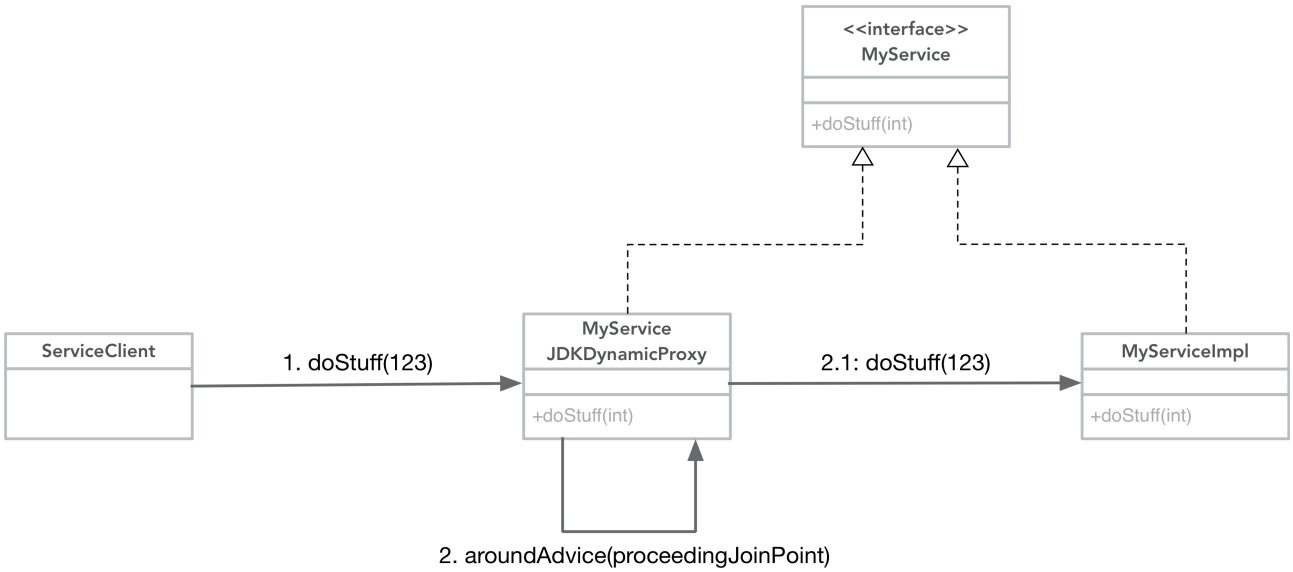
Spring AOP uses two slightly different types of proxy techniques:

- JDK dynamic proxies
- CGLIB proxies

JDK Dynamic Proxies

JDK dynamic proxies uses technology found in the Java runtime environment and thus require no additional libraries. Proxies are created at runtime by generating a class that implements all the interfaces that the target object implements.
Thus any methods found in the target object but not in any interface implemented by the target object cannot be proxied.

JDK dynamic proxies is the default proxy mechanism used by Spring AOP.



Client calling a service advised with an around advice implemented using a JDK dynamic proxy.

The above diagram combines an activity diagram, showing a call to a proxied Spring bean that has an around advice applied to it, and a class diagram, showing that both the generated proxy class and the service class *MyServiceImpl* implements the *MyService* interface. Note that the name of the proxy class will probably not be *MyServiceJDKDynamicProxy* when the proxy class is generated in a real application, but something less explaining.

CGLIB Proxies

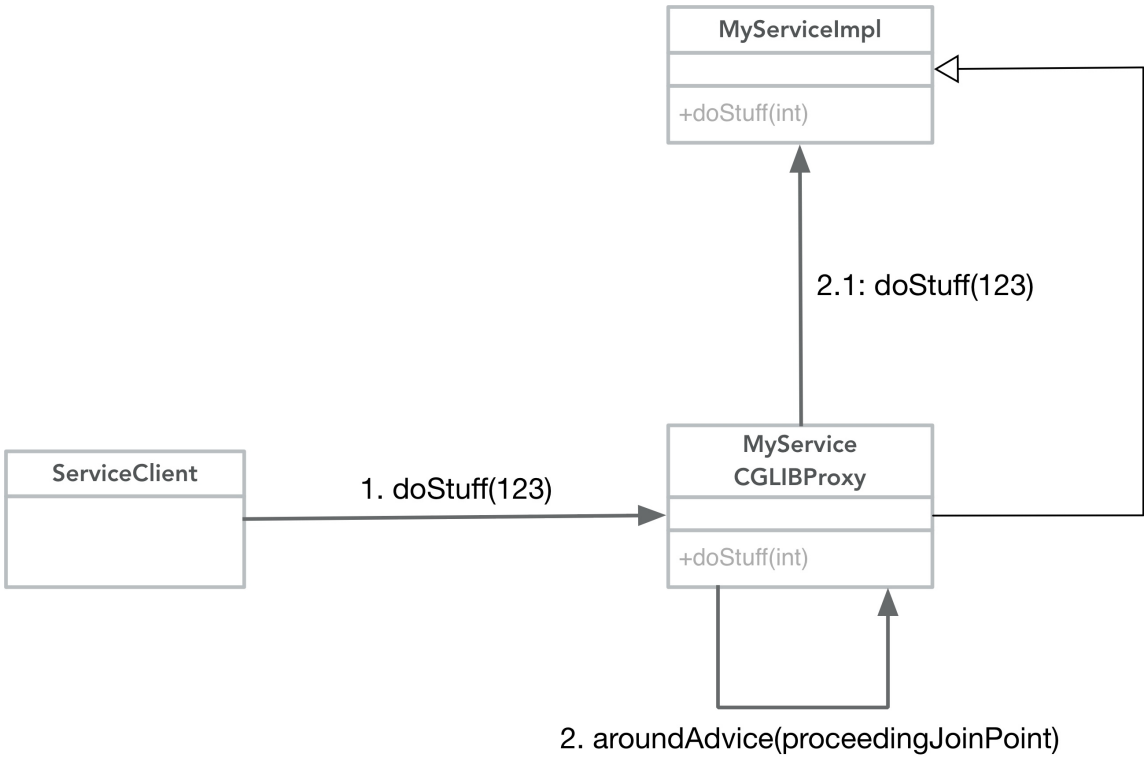
CGLIB is a third-party library used for, among other things, creating proxy objects at runtime. CGLIB proxies are created by generating a subclass of the class implementing the target object. This makes it possible to proxy not only methods implemented in interfaces, but in fact all public methods of the class.

The CGLIB proxy mechanism will be used by Spring AOP when the Spring bean for which to create a proxy does not implement any interfaces.

It is possible to instruct Spring AOP to use CGLIB proxies by default by setting the *proxyTargetClass* attribute of the *@EnableAspectJAutoProxy* annotation to true.

```
@EnableAspectJAutoProxy(proxyTargetClass = true)
```

Spring Java configuration classes, annotated with @Configuration, will always be proxied using CGLIB.



Client calling a service advised with an around advice implemented using a CGLIB proxy.

This is another diagram that combines an activity diagram, showing a call to a proxied Spring bean that has an around advice applied to it, and a class diagram, showing that the proxy class generated by CGLIB inherits from the *MyServiceImpl* class. Again, the name of the proxy class will probably be less informative in a real application.

References:

- Spring 5 Reference: Proxying mechanisms
- Spring 5 Reference: AOP concepts

- Spring 5 Reference: AOP Proxies
- Oracle: Java Technotes – Dynamic Proxy Classes
- CGLIB

Which are the limitations of the two proxy-types?

Each of the two techniques used by Spring AOP to create proxies have their limitations as described in the respective section below. There is one limitation common to both proxy types:

- Invocation of advised methods on self.
  A proxy implements the advice which is executed prior to invoking the method on a Spring bean. The Spring bean being proxied is not aware of the proxy and when a calling a method on itself, the proxy will not be invoked.

JDK Dynamic Proxies

Limitations of JDK dynamic proxies are:

- Class for which a proxy is to be created must implement an interface.
  This is due to the way that JDK dynamic proxies are implemented – see section on JDK dynamic proxies above!
- Only public methods in implemented interfaces will be proxied.
  Invocation of methods on self will never be proxied, see above limitation common to both proxy types. Thus only public methods declared in an implemented interface, which are the only type of method that can be invoked from other objects, will be proxied.

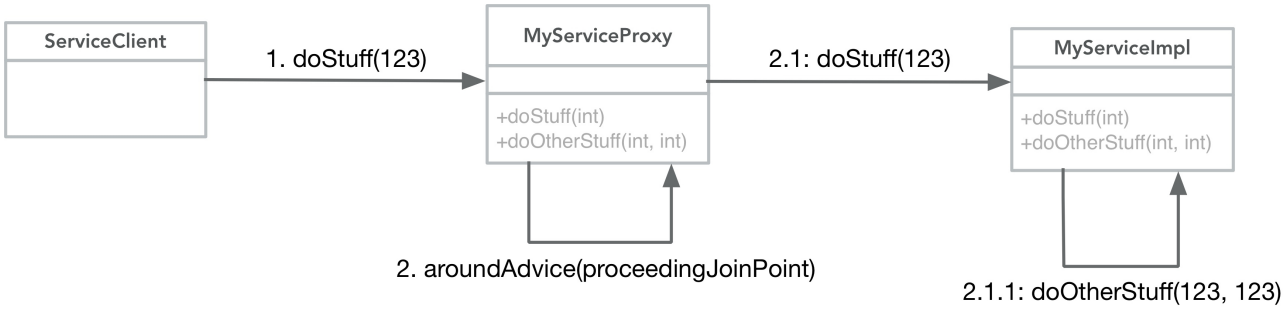
CGLIB Proxies

Limitations of CGLIB proxies are:

- Class for which a proxy is to be created must not be final.
  Since CGLIB creates a subclass of the class for which a proxy is to be created, the latter class must not be final since this prevents subclassing.
- Method(s) in class for which a proxy is to be created must not be final.
  The motivation is analogous to above; final methods cannot be overridden in subclasses.
- Only public and protected methods will be proxied.
  Invocation of methods on self will never be proxied, see above limitation common to both proxy types. See SPR-15354.

What visibility must Spring bean methods have to be proxied using Spring AOP?

A common limitation of both JDK dynamic proxies and CGLIB proxies is that neither of these two proxy methods support calls to methods that originate from the target object itself.



Proxied Spring bean invoking doOtherStuff method on itself.
The bean is not aware of the proxy and will thus bypass it entirely.

Such calls will bypass the proxy and invoke the original method directly, since the target object is not aware of being proxied.
Thus only public methods of Spring beans will be proxied with the additional limitation that the call to the public method must originate from outside of the Spring bean, in order for the proxy to intercept the call.

How many advice types does Spring support. Can you name each one?

Spring AOP supports the following types of advice:

- Before advice.
  Executed before a join point. Cannot prevent proceeding to the join point unless the advice throws an exception.
- After returning advice.
  Executed after execution of a join point completed without throwing exceptions.
- After throwing advice.
  Executed after execution of a join point that resulted in an exception being thrown.
- After (finally) advice.
  Executed after execution of a join point, regardless of whether an exception was thrown or not.
- Around advice.
  Executed before and after (around) a join point. Can chose to execute the join point or not. Can chose to return any return value from the execution of the join point or return some other return value.

References:

- Spring 5 Reference: AOP concepts

What are they used for?

The following are some examples of use-cases for the different types of advice listed above. The list of use-cases is not exhaustive.

Note: In the examples below, in-place pointcut expressions have been used for brevity.

Before Advice

Before advice will, as before, always proceed to the join point unless an execution is thrown from within the advice code. This makes it suitable for use-cases like:

- Access control (security)
  Authorization can be implemented using before advice, throwing an exception if the current user is not authorized.
- Statistics
  Counting the number of invocations of a join point.

Example:

```
    @Before("execution(* se..*.*Service.*(..))")
    public void beforeAdviceExample(final JoinPoint inJoinPoint) {

        final Object[] theArguments = inJoinPoint.getArgs();

        System.out.printf("** Before calling method: %s%n",
            inJoinPoint.getSignature().toShortString());

        System.out.println("   Arguments:");

        for (Object theArgument : theArguments) {

            System.out.printf("    - %s%n", theArgument.toString());

        }

    }
```

The before advice in the example above will be executed before any method in a class which name ends with “Service” that is located in the “se” package, or a sub-package, is invoked.
The advice parameter *inJoinPoint* of the type *JoinPoint* can be used to access information about the join point; in the example the signature and the arguments of the advised method are retrieved and printed to the console. Using a *JoinPoint* parameter in an advice is optional.

After Returning Advice

After returning advice will, as before, be invoked after the execution of a join point has completed without throwing any exceptions. Examples of use-cases are:

- Statistics
  Counting the number of successful invocations of a join point.
- Data validation
  Validating the data produced by the advised method.

Example:

```
    @AfterReturning(pointcut = "execution(* se..*.*Service.*(..))",
        returning = "inReturnValue")
    public void afterReturningAdviceExample(
        final JoinPoint inJoinPoint, final Object inReturnValue) {

        System.out.printf("** After returning from method: %s%n",
            inJoinPoint.getSignature().toShortString());

        System.out.printf("   Return value: %s%n", inReturnValue);

    }
```

The above advice will be executed after the successful completion of any method in a class which name ends with “Service” that is located in the “se” package, or a sub-package, with the return value of the method being passed to the advice in the *inReturnValue* parameter.

After Throwing Advice

After throwing advice will, as before, be invoked after the execution of a join point that resulted in an exception being thrown. Examples of use-cases are:

- Error handling
  Some examples of error handling that can be implemented using after throwing advice are:
  Saving additional information connected to an error.

Sending alerts when an error has occurred.
Attempt error recovery.

- Statistics
  Counting the number of invocations of a join point that resulted in an exception being thrown.
  Counting the number of exceptions of a certain type being thrown.

Example:

```
    @AfterThrowing(pointcut = "execution(* se..*.*Service.*(..))",
        throwing="inException")
    public void afterThrowingAdviceExample(
        final JoinPoint inJoinPoint, final Throwable inException) {

        System.out.printf("** After throwing from method: %s%n",
            inJoinPoint.getSignature().toShortString());

        System.out.printf("   Exception thrown: %s%n",
            inException.getClass().getSimpleName());

        System.out.printf("   Exception message: %s%n", inException.getMessage());

    }
```

The above advice will be executed if any method in a class which name ends with “Service” that is located in the “se” package, or a sub-package, throws an exception. The exception thrown will be passed to the advice in the *inException* parameter. The type of the *inExecption* parameter can be used to filter the exception types for which the advice will be executed.

After (Finally) Advice

After finally advice will, as before, be invoked after the execution of a join point regardless of whether the execution completed successfully or resulted in an exception being thrown. Possible use-cases for after finally advice are the same as for after returning and after throwing advice - please refer to the above sections! An additional use-case for after (finally) advice is:

- Releasing resources
  As with finally-blocks in try-finally, the after (finally) advice is always executed after the completion of the join point and can thus ensure that resources are always released.

Example:

```
    @After("execution(* se..*.*Service.*(..))")
    public void afterFinallyAdviceExample(final JoinPoint inJoinPoint) {

        System.out.printf("** After finally from method: %s%n",
            inJoinPoint.getSignature().toShortString());

    }
```

The above advice will be executed after any method in a class which name ends with “Service” that is located in the “se” package, or a sub-package, finished executing, regardless of whether the execution finished successfully or resulted in an exception being thrown.

Around Advice

Around advice is the most powerful of the advice types. The advice may select whether or not to execute the join point, catch an exception and throw another exception or not throw any exception at all etc etc. Around advice can be used for all of the use-cases for AOP.

Example:

```
    @Around("execution(* se..*.*Service.*(..))")
    public Object loggingAdvice(
        final ProceedingJoinPoint inProceedingJoinPoint) throws Throwable {
        System.out.printf("* Before service method '%s' invocation.%n",
            inProceedingJoinPoint.getSignature().toShortString());

        /* Invoke the join point. */
        final Object theResult = inProceedingJoinPoint.proceed();

        System.out.printf("* After service method '%s' invocation.%n",
            inProceedingJoinPoint.getSignature().toShortString());

        return theResult;
    }
```

The above advice will be executed before any method in a class which name ends with “Service” that is located in the “se” package, or a sub-package, is invoked. The advice-method declares a *Throwable* exception – this is due to that the method *proceed* in *PoceedingJoinPoint* declares this exception and that the above advice not catching that exception.
The *ProceedingJoinPoint* type is a specialization of the *JoinPoint* type we have seen earlier, offering the same facilities for retrieving information about the join point. The *ProceedingJoinPoint* interface adds two *proceed* methods for proceeding with the next advice or with the target method.

Which two advices can you use if you would like to try and catch exceptions?

Only around advice allows you to catch exceptions in an advice that occur during execution of a join point.

```
    @Around("publicServiceMethodInSpringPackagePointcut()")
    public Object loggingAdvice(
        final ProceedingJoinPoint inProceedingJoinPoint) throws Throwable {
        System.out.printf("* Before service method '%s' invocation.%n",
            inProceedingJoinPoint.getSignature().toShortString());
        Object theResult;

        try {
            /* Invoke the join point. */
```

```
            theResult = inProceedingJoinPoint.proceed();

            System.out.printf("* After service method '%s' invocation.%n",
                inProceedingJoinPoint.getSignature().toShortString());
        } catch (final Throwable theThrowable) {
            System.out.printf("* After service method '%s' invocation. "
                + "An exception occurred: %s with the message: %s%n",
                inProceedingJoinPoint.getSignature().toShortString(),
                theThrowable.getClass().getSimpleName(),
                theThrowable.getMessage());

            theResult = null;
        }

        return theResult;
    }
```

The above example implements an around advice that invokes the *proceed* method on the *ProceedingJoinPoint* inside a try-catch block. Exceptions occurring during execution of the join point will be caught and logged.

What do you have to do to enable the detection of the @Aspect annotation?

To enable detection of Spring beans implementing advice which implementation classes are annotated with the @Aspect annotation, the @EnableAspectJAutoProxy annotation should be applied to a @Configuration class. When using the @EnableAspectJAutoProxy annotation, the aspectjweaver.jar library from AspectJ needs to be on the classpath.

In order for advice to be created from classes annotated with the @Aspect annotation, Spring beans need to be created from these classes. This can be accomplished in two ways:

- Annotate advice classes with @Component.
  Thus advice classes should be annotated with both @Aspect and @Component. With component scanning enabled, advice classes will be auto-detected.
- Create Spring beans from the advice classes.
  Use a regular method annotated with @Bean in a @Configuration class.

What does @EnableAspectJAutoProxy do?

The @EnableAspectJAutoProxy annotation enables support for handling Spring beans that are annotated with AspectJ's @Aspect annotation. It is important to note that the @Aspect annotation in itself does not cause any Spring beans to be created from the annotated classes – see above!

References:

- Spring 5 Reference: Enabling @AspectJ Support
- Spring 5 Reference: Declaring an aspect
- Spring 5 API Documentation: @EnableAspectJAutoProxy

If shown pointcut expressions, would you understand them?

Basic Structure of Pointcut Expressions

The basic structure of a pointcut expression consists of two parts; a pointcut designator and an pattern that selects join points of the type determined by the pointcut designator.
Example:

```
execution(* se..*.*Service.*(..))
```

In the above example the pointcut designator is “execution” and the pattern specifying which execution joinpoints to select is “* se..*.*Service.*(..)”.

Combining Pointcut Expressions

The above expression consists of a single pointcut designator. It is possible to combine multiple pointcut designators into one single pointcut expression, as shown in the following example:

```
execution(public * *(..)) && within(se.ivankrizsan.spring..*)
```

The pointcut in this example will select all join points that are:

1. Execution of public methods within all packages that have any return type and take zero or more parameters.
2. AND
3. Within the package se.ivankrizsan.spring or any subpackage of this package.

To combine the two pointcut expressions in the example into one, the && (AND) operator is used which results in only join points that are selected by both sub-expressions will be selected.
The operators that can be used when combining pointcuts are:

- !
  Not – negates the pointcut expression.
  Example: !within(se.ivankrizsan.spring..*) will result in join points that are NOT within the package se.ivankrizsan.spring or any of its subpackages being selected.
- &&
  And – logical and of pointcut expressions.
  As earlier, this will result in join points being selected by both the pointcut expressions (on the left and right sides of the && operator) being selected by the combined pointcut expression.
- ||
  Or – logical or of pointcut expressions.
  Join points being selected by at least one of the two pointcut expressions on either side of the operator will be selected by the combined pointcut.

Pointcut Designators

The following sections describe the pointcut designators supported by Spring AOP. If using an unsupported AspectJ pointcut designator with Spring AOP, an *IllegalArgumentException* will be thrown.

Pointcut designator: execution

The *execution* pointcut designator matches method execution join points. This is the most commonly used pointcut designator in Spring AOP.

Example:

```
execution(public String se.ivankrizsan.spring.aopexamples.MySuperServiceImpl.*(String))
```

The pattern specifying which method execution join points to select consists of the following parts:

```
[method visibility] [return type] [package].[class].[method]([parameters] [throws
exceptions])
```

- Method visibility
  Can be one of private, protected, public. Can be negated using !. May be omitted, in which case all method visibilities will match.
- Return type
  Object or primitive type. Can be negated with !. Wildcard * can be used, in which case all return types will be matched.
- Package
  Package in which class(es) is/are located. May be omitted. Wildcard ".." may be used last in package name to include all sub-packages. Wildcard * may be used in package name.
- Class
  Class in which method(s) to be selected is/are located. May be omitted. Includes subclasses of the specified class. Wildcard * may be used.
- Method
  Name of method(s) in which join points to be selected are located. Whole or partial method name. Wildcard * may be used, for example "gree*" will match a method named *greet* and any other methods which names start with "gree".
- Parameters
  Object or primitive types of parameters. A parameter-type can be negated with !, that is !int will match any type except for int. The wildcard ".." can be used to match zero or more subsequent parameters.
- Exceptions
  Type(s) of exception(s) that matching method(s) throws. An exception type can be negated using !.

Pointcut designator: within

The *within* pointcut designator matches join points located in one or more classes, optionally specifying the package(s) in which the class(es) is/are located.

Example:

```
within(se..MySuperServiceImpl)
```

The pattern specifying which join points to select consists of the following parts:

```
[package].[class]
```

- Package
  Package in which class(es) to be selected is/are located. May be omitted. Wildcard ".." may be used last in package name to include all sub-packages. Wildcard * may be used in package name.
- Class
  Class(es) in which join points are to be selected. Wildcard * may be used. Join points in subclasses of the specified class will also be matched.

Pointcut designator: this

The *this* pointcut designator matches all join points where the currently executing object is of specified type (class or interface). With Spring AOP, this will be the proxy object.

Example:

```
this(MySuperService)
```

In this example, *MySuperService* is an interface. The above pointcut expression will match join points in proxy objects that implement the *MySuperService* interface.

The pattern specifying which join points to select only consists of a single type. Wildcards can not be used in type names.

Pointcut designator: target

The *target* pointcut designator matches all join point where the target object, for instance the object on which a method call is being made, is of specified type (class or interface). With Spring AOP, the target object will reference the Spring bean being proxied.

Example:

```
target(MySuperServiceImpl)
```

The pattern specifying which join points to select only consists of a single type. Wildcards can not be used in type names.

Pointcut designator: args

The *args* pointcut designator matches join points, in Spring AOP method execution, where the argument(s) are of the specified type(s).

Example:

```
args(long, long)
```

The above example selects join points where the arguments are two long integers.

The .. wildcard may be used to specify zero or more parameters of arbitrary type.
The * wildcard can be used to specify one parameter of any type.
Package information may be included in the pattern specifying which join points to select. The following example selects all join points where the arguments are of any type from the java.util package:

```
args(java.util.*)
```

Pointcut designator: @target

The @target pointcut designator matches join points in classes annotated with the specified annotation(s).

Example:

```
@target(org.springframework.stereotype.Service)
```

The above example selects all join points in all classes annotated with the Spring @Service annotation.

Pointcut designator: @args

The @args pointcut designator matches join points where an argument type (class) is annotated with the specified annotation. Note that it is not the argument that is to be annotated, but the class.

Let’s look at a more extensive example to make it more clear:

Given the @CleanData annotation and the *Customer* class:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface CleanData {
}
```

```
@CleanData
public class Customer {
    protected String firstName;
    protected String lastName;
    …
}
```

...and a method in a service class:

```
    public void processCustomer(final Customer inCustomer) {
        /* Do stuff with the customer. */
    }
```

Finally, a before-advice that uses the @args pointcut designator:

```
@Before("@args(se.ivankrizsan.spring.aopexamples.CleanData) &&
within(se.ivankrizsan.spring..*)")
    public void testAtArgs() {
        System.out.println("Got a service parameter annotated with @CleanData!");
    }
```

Note that there are two pointcut designators in the @Before annotation. The *within* pointcut designator is used to narrow the scope of the selected join points. If not present, Spring AOP will attempt to create CGLIB proxies of final classes, which will fail with an error message similar to this:

```
java.lang.IllegalArgumentException: Cannot subclass final class
org.springframework.boot.autoconfigure.AutoConfigurationPackages$BasePackages
```

Pointcut designator: @within

The @within pointcut designator matches join points in classes annotated with specified annotation.

Example:

```
@within(org.springframework.stereotype.Service)
```

The above pointcut will select all join points in all classes annotated with the Spring @Service annotation.

Pointcut designator: @annotation

The @annotation pointcut designator matches join points in methods annotated with specified annotation.

Example:

```
@annotation(se.ivankrizsan.spring.aopexamples.MySuperSecurityAnnotation)
```

The above pointcut will select all join points in all methods annotated with the @MySuperSecurity Annotation annotation in all classes.

Pointcut designator: bean

While all of the above pointcut designators are supported in AspectJ, the *bean* pointcut designator is not. This pointcut designator selects all join points within a Spring bean.

Example:

```
bean(mySuperService)
```

The above pointcut will select all join points in the Spring bean named “mySuperService”.

The pattern specifying which join points to select consists of the name or id of a Spring bean. The wildcard * may be used in the pattern, making it possible to match a set of Spring beans with one pointcut expression.

References:

- Spring 5 Reference: Declaring a pointcut
- The AspectJ Programming Guide: Quick Reference: Pointcuts
- The AspectJ Programming Guide: Language Semantics: Pointcuts
- The AspectJ Developer's Notebook: Join Point Signatures
- Spring 5 Reference: Supported Pointcut Designators
- The AspectJ Programming Guide: Pattern Summary

For example, in the course we matched getter methods on Spring Beans, what would be the correct pointcut expression to match both getter and setter methods?

The following pointcut expression matches all getter and setter methods in the entire codebase of the application. When used in your own application, you may want to limit the scope to, for instance, the getter and setter methods of the entity classes using a *within* pointcut expression.

```
execution(void set*(*)) || execution(* get*())
```

The above pointcut expression combines two pointcut expressions that selects joinpoints as follows:

- Method execution of methods which name starts with "set" and that return void and takes one single parameter.
- Method execution of methods which name starts with "get" and that returns arbitrary type and takes no parameters.

Since OR (||) is used between the two pointcut expressions, the resulting set of join points selected will be the union of the sets of join points selected by the two sub-expressions.

What is the JoinPoint argument used for?

As seen in earlier examples, a parameter of the type *JoinPoint* can be added to methods implementing the following types of advice:

- Before
- After returning
- After throwing
- After (finally)

The parameter must, if present, be the first parameter of the advice method.

When the advice is invoked, the parameter will hold a reference to an object that holds static information about the join point as well as state information.

Examples of static information:

- Kind (type) of join point
  With Spring AOP, this will always be method execution.
- Signature at the join point
  With Spring AOP, this will be the signature of the advised method.

Examples of dynamic information available from the *JoinPoint* object:

- Target object
  With Spring AOP this will always be a Spring bean that contains the advised method.
  Will always be the same as the *target* pointcut designator.
- Currently executing object
  Should return the proxy object according to the Spring reference documentation.
  Should always be the same as the *this* pointcut designator.

The Spring reference documentation states that the *JoinPoint.getTarget* method should return the target object, which I interpret as the Spring bean being proxied. In addition the documentation states that the *JoinPoint.getThis* method should return the proxy object.
However, in an example program I developed to examine the behavior of Spring AOP, both the *getTarget* and *getThis* methods always returned one and the same object, which was the target object.

With AspectJ, the references held in this and target depend on the type of pointcut:
When using a *call* pointcut, the interception takes place in the caller. This results in the reference in this and the reference in the target pointing at different objects.
When using an *execution* pointcut, the interception takes place in the target object on which the method to be executed is implemented. This results in the reference in this and the reference in the

target pointing at one and the same object.

References:

- AspectJ API Documentation: JoinPoint
- AspectJ API Documentation: JoinPoint.StaticPart
- Spring 5 Reference: Advice parameters
- AspectJ Reference: Pointcuts

What is a ProceedingJoinPoint? When is it used?

The following example, also seen earlier, shows the use of the *ProceedingJoinPoint* class as a parameter to an around advice. This type is used as the first parameter of a method implementing an around advice.

```
    @Around("publicServiceMethodInSpringPackagePointcut()")
    public Object loggingAdvice(
        final ProceedingJoinPoint inProceedingJoinPoint) throws Throwable {
        System.out.printf("* Before service method '%s' invocation.%n",
            inProceedingJoinPoint.getSignature().toShortString());

        /* Invoke the join point. */
        final Object theResult = inProceedingJoinPoint.proceed();

        System.out.printf("* After service method '%s' invocation.%n",
            inProceedingJoinPoint.getSignature().toShortString());

        return theResult;
    }
```

While it is possible to implement an around advice method without a *ProceedingJoinPoint* parameter, it will be impossible to invoke the next advice method or target method in such a case.

Since the *ProceedingJoinPoint* class is a subclass of the *JoinPoint* class it contains all the information described in the above section on the *JoinPoint* class. In addition, the *ProceedingJoinPoint* class contains these two additional methods:

- proceed()
  Proceed to invoke the next advice method or the target method without passing any parameters. Will return an *Object*. May throw a *Throwable* exception.
- proceed(Object[] args)
  Proceed to invoke the next advice method or the target method passing an array of objects as argument to the method to be invoked. If this() and/or target() was used in the pointcut for binding, this() must be the first element in the array of arguments and target() must be the second (first, if this() not used) element in the array of arguments.
  Will return an *Object*. May throw a *Throwable* exception.

References:

- AspectJ API Documentation: ProceedingJoinPoint
- Spring 5 Reference: Around advice

Chapter 3

Data Management: JDBC, Transactions, JPA, Spring Data

What is the difference between checked and unchecked exceptions?

Checked exceptions are exceptions that the Java compiler requires to be declared in the signature of methods that throw this type of exceptions. If a method calls another method that declares one or more checked exceptions in its method signature, the calling method must either catch these exceptions or declare the exceptions in its method signature.
The class *java.lang.Exception* and its subclasses, except for *java.lang.RuntimeException* and any subclass of *RuntimeException*, are checked exceptions.

Unchecked exceptions are exceptions that the Java compiler does not require to be declared in the signature of methods or to be caught in methods invoking other methods that may throw unchecked exceptions.

Why does Spring prefer unchecked exceptions?

Checked exceptions forces developers to either implement error handling in the form of try-catch blocks or to declare exceptions thrown by underlying methods in the method signature.
This can result in cluttered code and/or unnecessary coupling to the underlying methods.

Unchecked exceptions gives developers the freedom of choice as to decide where to implement error handling and removes any coupling related to exceptions.

What is the data access exception hierarchy?

The data access exception hierarchy is the *DataAccessException* class and all of its subclasses in the Spring Framework. All the exceptions in this exception hierarchy are unchecked.
The purpose of the data access exception hierarchy is isolate application developers from the particulars of JDBC data access APIs, for instance database drivers from different vendors. This in turn enables easier switching between different JDBC data access APIs.

References:

- Java SE 8 API Documentation: Throwable
- Java SE 8 API Documentation: Error
- Java SE 8 API Documentation: Exception
- Java SE 8 API Documentation: RuntimeException
- Spring 5 API Documentation: DataAccessException

How do you configure a DataSource in Spring? Which bean is very useful for development/test databases?

The *javax.sql.DataSource* interface is the interface from which all data-source classes related to SQL stem. The core Spring Framework contain the following root data-source classes and interfaces that all implement this interface:

- DelegatingDataSource
- AbstractDataSource
- SmartDataSource
- EmbeddedDatabase

How do you configure a DataSource in Spring?

Obtaining a *DataSource* in a Spring application depends on whether the application is deployed to an application or web server, for example GlassFish or Apache Tomcat, or if the application is a standalone application.

DataSource in a standalone application

After having chosen the appropriate *DataSource* implementation class, a data-source bean is created like any other bean. The following example creates a data-source that retrieves data from a HSQL database using the Apache Commons DBCP *BasicDataSource* data-source:

```
    @Bean
    public DataSource dataSource() {
        final BasicDataSource theDataSource = new BasicDataSource();
        theDataSource.setDriverClassName("org.hsqldb.jdbcDriver");
        theDataSource.setUrl("jdbc:hsqldb:hsql://localhost:1234/mydatabase");
        theDataSource.setUsername("ivan");
        theDataSource.setPassword("secret");
        return theDataSource;
    }
```

If the application uses Spring Boot, then it is not necessary to create a *DataSource* bean. Setting the values of a few properties are sufficient:

```
spring.datasource.url= jdbc:hsqldb:hsql://localhost:1234/mydatabase
spring.datasource.username=ivan
spring.datasource.password=secret
```

DataSource in an application deployed to a server

If the application is deployed to an application server then a way to obtain a data-source is by performing a JNDI lookup like in this example:

```
    @Bean
    public DataSource dataSource() {
        final JndiDataSourceLookup theDataSourceLookup = new JndiDataSourceLookup();
        final DataSource theDataSource =
            theDataSourceLookup.getDataSource("java:comp/env/jdbc/MyDatabase");
        return theDataSource;
    }
```

Spring Boot applications need only to rely on setting one single property:

```
spring.datasource.jndi-name=java:comp/env/jdbc/MyDatabase
```

References:

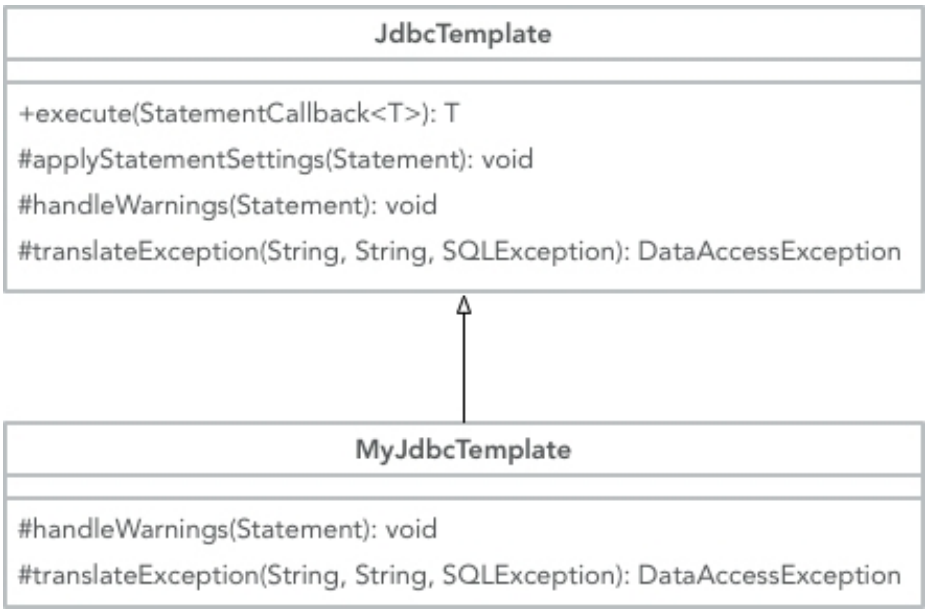
- Spring 5 API Documentation: AbstractDataSource
- Spring 5 API Documentation: DelegatingDataSource
- Spring 5 API Documentation: SmartDataSource
- Spring 5 API Documentation: EmbeddedDatabase
- Apache Commons DBCP
- Spring 5 API Documentation: JndiDataSourceLookup
- Oracle JavaSE 8 Documentation: Java Naming and Directory Interface (JNDI)
- Spring Boot 2 Reference: Connection to a JNDI DataSource

What is the Template design pattern and what is the JDBC template?

What is the Template Design Pattern

The template (method) design pattern is a design pattern in which an algorithm is defined and the steps of the algorithm are refactored to methods with protected visibility. The class defining the algorithm may provide abstract methods for the different steps of the algorithm, letting subclasses define all steps. Alternatively the class defining the algorithm may define default implementations of the different steps of the algorithm, allowing subclasses to customize only selected methods as desired.

Example:



The figure above shows the Spring *JdbcTemplate* class and selected methods. The execute method shown defines an algorithm that executes a SQL statement which consists of the customizable steps implemented in the *applyStatementSettings*, *handleWarnings* and *translateException* methods that all have protected visibility. The class *MyJdbcTemplate* inherits from *JdbcTemplate* and customizes the behavior of the *handleWarnings* and *translateException* methods.

What is the JDBC template?

The Spring *JdbcTemplate* class is a Spring class that simplifies the use of JDBC by implementing common workflows for querying, updating, statement execution etc. Some benefits of using the *JdbcTemplate* class are:

- Simplification
  Reduces the amount of (boilerplate) code necessary to perform JDBC operations.
- Handles exceptions
  Exceptions are properly handled ensuring that resources are closed or released.
- Translates exceptions
  Exceptions are translated to the appropriate exception in the *DataAccessException* hierarchy (unchecked exceptions) which are also vendor-agnostic.
- Avoids common mistakes
  For example, ensures that statements are property closed and connections are released after having performed JDBC operations.
- Allows for customization of core functionality.
  An example is exception translation. See the above section on the template design pattern.
- Allows for customization of per-use functionality.
  An example is mapping of rows in a result set to a Java object. This is accomplished using callbacks. Please refer to the next section for further details.

Instances of *JdbcTemplate* are thread-safe after they have been created and configured.

References:

- Wikipedia: Template method pattern
- Wikipedia: Callback
- Spring 5 API Documentation: JdbcTemplate

What is a callback? What are the three JdbcTemplate callback interfaces that can be used with queries? What is each used for? (You would not have to remember the interface names in the exam, but you should know what they do if you see them in a code sample)

What is a callback?

A callback is code or reference to a piece of code that is passed as an argument to a method that, at some point during the execution of the methods, will call the code passed as an argument.
In Java a callback can be a reference to a Java object that implements a certain interface or, starting with Java 8, a lambda expression.

What are the three JdbcTemplate callback interfaces that can be used with queries? What is each used for?

The three callback interfaces that can be used with queries to extract result data are:

- ResultSetExtractor
  Allows for processing of an entire result set, possibly consisting multiple rows of data, at once. Suitable when more than a single row of data is needed to create a Java object holding query result data. Result set extractors are typically stateless. Note that the *extractData* method in this interface returns a Java object.

- RowCallbackHandler
  Allows for processing rows in a result set one by one typically accumulating some type of result. Row callback handlers are typically stateful, storing the accumulated result in an instance variable. Note that the *processRow* method in this interface has a void return type.

- RowMapper
  Allows for processing rows in a result set one by one and creating a Java object for each row. Row mappers are typically stateless. Note that the *mapRow* method in this interface returns a Java object.

References:

- Wikipedia: Callback
- Wikipedia: Anonymous function (Lambda)
- Spring 5 API Documentation: ResultSetExtractor
- Spring 5 API Documentation: RowCallbackHandler
- Spring 5 API Documentation: RowMapper

Can you execute a plain SQL statement with the JDBC template?

Plain SQL statements can be executed using the *JdbcTemplate* class. The following methods accept one or more SQL strings as parameters. Note that there may be multiple versions of a method that take different parameters.

- batchUpdate
- execute
- query
- queryForList
- queryForMap
- queryForObject
- queryForRowSet
- update

References:

- Spring 5 API Documentation: JdbcTemplate

When does the JDBC template acquire (and release) a connection - for every method called or once per template? Why?

JdbcTemplate acquire and release a database connection for every method called. That is, a connection is acquired immediately before executing the operation at hand and released immediately after the operation has completed, be it successfully or with an exception thrown.

The reason for this is to avoid holding on to resources (database connections) longer than necessary and creating as few database connections as possible, since creating connections can be a potentially expensive operation. When database connection pooling is used connections are returned to the pool for others to use.

How does the JdbcTemplate support generic queries? How does it return objects and lists/maps of objects?

JdbcTemplate contains seven different *queryForList* methods and three different *queryForMap* methods.

The *queryForList* methods all return a list containing the resulting rows of the query and comes in two flavors:

- One type that takes an element type as parameter.
  This type of *queryForList* method returns a list containing objects of the type specified by the element type parameter.
- The other type that do not have any element type parameter.
  This type of *queryForList* method returns a list containing maps with string keys and *Object* values. Each map contains a row of the query result with the column names as keys and the column values as values in the map.

All *queryForMap* methods are expected to return one single row. The resulting row is returned in a map with string keys and *Object* values. Column names are the keys of the map entries and the column values are the values.

References:

- Spring 5 API Documentation: JdbcTemplate

What is a transaction? What is the difference between a local and a global transaction?

What is a transaction?

A transaction is an operation that consists of a number of tasks that takes place as a single unit – either all tasks are performed or no tasks are performed. If a task that is part of a transaction do not complete successfully, the other tasks in the transaction will either not be performed or, for tasks that have already been performed, be reverted.

A reliable transaction system enforces the ACID principle:

- Atomicity
  The changes within a transaction are either all applied or none applied.
  “All or nothing”
- Consistency
  Any integrity constraints, for instance of a database, are not violated.
- Isolation
  Transactions are isolated from each other and do not affect each other.
- Durability
  Changes applied as the result of a successfully completed transaction are durable.

What is the difference between a local and a global transaction?

Global transactions allow for transactions to span multiple transactional resources. As an example consider a global transaction that spans a database update operation and the posting of a message to the queue of a message broker. If the database operation succeeds but the posting to the queue fails then the database operation will be rolled back (undone). Similarly if posting to the queue succeeds but the database operation fails, the message posted to the queue will be rolled back and will thus not appear on the queue. Not until both operations succeed will the database update come into effect and a message be made available for consumption on the queue.
Note that a transaction that is to span operations on two different databases needs to be a global transaction.

Local transactions are transactions associated with one single resource, such as one single database or a queue of a message broker, but not both in one and the same transaction.

References:

- Wikipedia: Transaction processing
- Wikipedia: Distributed transaction
- Spring 5 Reference: Advantages of the Spring Framework’s transaction support model

Is a transaction a cross cutting concern? How is it implemented by Spring?

As already mentioned in the section on Aspect Oriented Programming, transaction management is a cross-cutting concern. In the Spring framework declarative transaction management is implemented using Spring AOP.

References:

- Spring 5 Reference: Understanding the Spring Framework’s declarative transaction implementation

How are you going to define a transaction in Spring?

The following two steps are all required to use Spring transaction management in a Spring application:

- Declare a *PlatformTransactionManager* bean.
  Choose a class implementing this interface that supplies transaction management for the transactional resource(s) that are to be used. Some examples are *JmsTransactionManager* (for a single JMS connection factory), *JpaTransactionManager* (for a single JPA entity manager factory).
- If using annotation-driven transaction management, then apply the @EnableTransactionManagement annotation to exactly one @Configuration class in the application.
- Declare transaction boundaries in the application code.
  This can be accomplished using one or more of the following:
  @Transactional annotation
  Spring XML configuration (not in the scope of this book)
  Programmatic transaction management

What does @Transactional do? What is the PlatformTransactionManager ?

@Transactional

The @Transactional annotation is used for declarative transaction management and can be applied to methods and classes. This annotation is used to specify the transaction attributes for the method which it annotates or, if applied on class level, all the methods in the class.

The following can be configured in the @Transactional annotation:

- isolation
  The transaction isolation level.
- noRollbackFor
  Exception class(es) that never are to cause a transaction rollback.
- noRollbackForClassName
  Names of exception class(es) that never are to cause a transaction rollback.
- propagation
  Transaction propagation.
- readOnly
  Set to true if transaction is a read-only transaction.

- rollbackFor
  Exception class(es) that are to cause a transaction rollback.
- rollbackForClassName
  Name of exception class(es) that are to cause a transaction rollback.
- timeout
  Transaction timeout.
- transactionManager
  Name of transaction manager Spring bean.

Spring allows for using the JPA *javax.transaction.Transactional* annotation as a replacement for the Spring @Transactional annotation, though it does not have as many configuration options.

What is the PlatformTransactionManager?

PlatformTransactionManager is the base interface for all transaction managers that can be used in the Spring framework's transaction infrastructure. Transaction managers (implementing this interface) can be used directly by applications, but it is recommended to use declarative transactions or the *TransactionTemplate* class.

The *PlatformTransactionManager* interface contain the following methods:

- void commit(TransactionStatus)
  Commits or rolls back the transaction related to the *TransactionStatus* object transaction depending on its status.
- void rollback(TransactionStatus)
  Rolls back the transaction related to the *TransactionStatus* object.
- TransactionStatus getTransaction(TransactionDefinition)
  Creates a new transaction and return it or return the currently active transaction, depending on how transaction propagation has been configured.

References:

- Spring 5 Reference: Declarative transaction management
- Spring 5 Reference: Programmatic transaction management
- Spring 5 API Documentation: Transactional
- Spring 5 API Documentation: PlatformTransactionManager
- Spring 5 API Documentation: JmsTransactionManager
- Spring 5 API Documentation: JpaTransactionManager
- Spring 5 API Documentation: TransactionTemplate

- Spring 5 API Documentation: EnableTransactionManagement

Is the JDBC template able to participate in an existing transaction?

Yes, the *JdbcTemplate* is able to participate in existing transactions both when declarative and programmatic transaction management is used. This is accomplished by wrapping the *DataSource* using a *TransactionAwareDataSourceProxy*.

References:

- Spring 5 Reference: TransactionAwareDataSourceProxy
- Spring 5 API Documentation: TransactionAwareDataSourceProxy

What is a transaction isolation level? How many do we have and how are they ordered?

What is a transaction isolation level?

Transaction isolation in database systems determine how the changes within a transaction are visible to other users and systems accessing the database prior to the transaction being committed. A higher isolation level reduces, or even eliminates, the chance for the problems described below that may appear when the database is updated and accessed concurrently. The drawback of higher isolation levels is a reduction of the ability of multiple users and systems concurrently accessing the database as well as increased use of system resources on the database server.

How many do we have and how are they ordered?

There are four isolation levels, here listed in descending order:

- Serializable
- Repeatable reads
- Read committed
- Read uncommitted

Serializable

Serializable is the highest isolation level and involves read and write locks on data associated with the processing in the transaction as well as range-locks if there are select queries with where-clauses. The locks are held until the end of the transaction.

Repeatable Reads

In the repeatable reads isolation level, read and write locks on data associated with the processing in the transaction are held until the end of the transaction. Range-locks are not used, so phantom reads can occur.

Example of a phantom read:

1. Transaction A queries table T for multiple rows of data.
   A where-clause is used to select the data, for instance all persons with an age in the range 20 to 50 years.
2. Transaction B inserts data into table T.
   Example: A new row is inserted that contains data for a person of the age 35.
3. Transaction A queries table T for data again.
   The same query as in step 1 is used, again selecting persons in the age-range 20 to 50 years.
   The result includes the data inserted into table T by transaction B, a person with the age 35.
   The phantom read has occurred.

The serializable isolation level will lock the entire range, age 20 to 50, for both reading and writing by other transactions – a so-called range-lock.

Read Committed

In the read committed isolation level, write locks on data associated with the processing in the transaction are held until the end of the transaction. Read locks are held, but only until the select-statement with which the read lock is associated has completed.

At this isolation level, phantom reads and non-repeatable reads can occur.

Example of a non-repeatable read:

1. Transaction A queries table T for data.
   As example, the data for a person, including the address of the person, with a certain name is queried.
2. Transaction B updates the data in table T that was read by transaction A and the transaction B is committed.
   As example, the address of the person which data was queried for by transaction A was updated.
3. Transaction A queries table T for data.
   If the data for the person retrieved in step 1 is once again read, transaction A will not see the same address as in step 1, but instead the updated address written in step 2.

Read Uncommitted

The read uncommitted isolation level is the lowest isolation level. At this isolation level, dirty reads may occur.

Example of a dirty read:

1. Transaction A queries table T for data.
   As example, the data for a person, including the address of the person, with a certain name is queried.
2. Transaction B updates the data in table T that was read by transaction A.
   Note that transaction B is need not be committed in order for the dirty read to occur.
   As example, the address of the person which data was queried for by transaction A was updated.
3. Transaction A queries table T for data.
   If the data for the person retrieved in step 1 is once again read, transaction A will not see the same address as in step 1, but instead the updated address written in step 2.

References:

- Wikipedia: Isolation (database systems)
- Wikipedia: Phantom reads

What is @EnableTransactionManagement for?

The @EnableTransactionManagement annotation is to annotate exactly one configuration class in an application in order to enable annotation-driven transaction management using the @Transactional annotation.

Components registered when the @EnableTransactionManagement annotation is used are:

- A *TransactionInterceptor*.
  Intercepts calls to @Transactional methods creating new transactions as necessary etc.
- A JDK Proxy or AspectJ advice.
  This advice intercepts methods annotated with @Transactional (or methods that are located in a class annotated with @Transactional).

The @EnableTransactionmanagement annotation have the following three optional elements:

- mode
  Allows for selecting the type of advice that should be used with transactions. Possible values are *AdviceMode.ASPECTJ* and *AdviceMode.PROXY* with the latter being the default.
- order
  Precedence of the transaction advice when more than one advice is applied to a join-point. Default value is *Ordered.LOWEST_PRECEDENCE*.
- proxyTargetClass
  True if CGLIB proxies are to be used, false if JDK interface-based proxies are to be used in the application (affects proxies for all Spring managed beans in the application!). Applicable only if the mode element is *AdviceMode.PROXY*.

References:

- Spring 5 API Documentation: EnableTransactionManagement
- Spring 5 API Documentation: TransactionInterceptor
- Spring 5 API Documentation: AdviceMode

What does transaction propagation mean?

Transaction propagation determines the way an existing transaction is used, depending on the transaction propagation configured in the @Transactional annotation on the method, when the method is invoked.

There are seven different options available when setting the propagation in a @Transactional annotation, all defined in the *Propagation* enumeration:

- MANDATORY
  There must be an existing transaction when the method is invoked, or an exception will be thrown.
- NESTED
  Executes in a nested transaction if a transaction exists, otherwise a new transaction will be created. This transaction propagation mode is not implemented in all transaction managers.
- NEVER
  Method is executed outside of a transaction. Throws exception if a transaction exists.
- NOT_SUPPORTED
  Method is executed outside of a transaction. Suspends any existing transaction.
- REQUIRED
  Method will be executed in the current transaction. If no transaction exists, one will be created.
- REQUIRES_NEW
  Creates a new transaction in which the method will be executed. Suspends any existing transaction-
- SUPPORTS
  Method will be executed in the current transaction, if one exists, or outside of a transaction if one does not exist.

References:

- Spring 5 Reference: Transaction propagation
- Spring 5 API Documentation: Propagation

What happens if one @Transactional annotated method is calling another @Transactional annotated method on the same object instance?

As described in the section on Aspect Oriented Programming discussing limitations of the Spring proxy-types, a self-invocation of a proxied Spring bean effectively bypasses the proxy and thus also any transaction interceptor managing transactions. Thus the second method, the method being invoked from another method in the bean, will execute in the same transaction context as the first. Any configuration in a @Transactional annotation on the second method will not come into effect.

If Spring transaction management is used with AspectJ, then any transaction-configuration using @Transactional on non-public methods will be honored.

References:

- Spring 5 Reference: Using @Transactional

Where can the @Transactional annotation be used? What is a typical usage if you put it at class level?

The @Transactional annotation can be used on class and method level both in classes and interfaces. When using Spring AOP proxies, only @Transactional annotations on public methods will have any effect – applying the @Transactional annotation to protected or private methods or methods with package visibility will not cause errors but will not give the desired transaction management, as above.

References:

- Spring 5 Reference: Using @Transactional

What does declarative transaction management mean?

Declarative transaction management means that the methods which need to be executed in the context of a transaction and the transaction properties for these methods are declared, as opposed to implemented. This is accomplished using annotations or Spring XML configuration.

References:

- Spring 5 Reference: Declarative transaction management

What is the default rollback policy? How can you override it?

The default rollback policy of Spring transaction management is that automatic rollback only takes place in the case of an unchecked exception being thrown.
The types of exceptions that are to cause a rollback can be configured using the *rollbackFor* element of the @Transactional annotation. In addition, the types of exceptions that not are to cause rollbacks can also be configured using the *noRollbackFor* element.

References:

- Spring 5 Reference: Declarative transaction management
- Spring 5 API Documentation: Transactional

What is the default rollback policy in a JUnit test, when you use the @RunWith(SpringJUnit4ClassRunner.class) in JUnit 4 or @ExtendWith(SpringExtension.class) in JUnit 5, and annotate your @Test annotated method with @Transactional ?

If a test-method annotated with @Test is also annotated with @Transactional, then the test-method will be executed in a transaction. Such a transaction will automatically be rolled back after the completion of the test-method. The reason for this is so that test-methods should be able to either modify the state of any database themselves or invoke methods that modifies the state of the database and have such changes reverted after the completion of the test method.

The rollback policy of a test can be changed using the @Rollback annotation and setting the value to false.

References:

- Spring 5 API Documentation: Rollback
- Spring 5 Reference: Testing – Transaction rollback and commit behaviour

Why is the term "unit of work" so important and why does JDBC AutoCommit violate this pattern?

The unit of work describes the atomicity characteristic of transactions. As earlier, it is either “all or nothing”, that is all operations that are transactional and take place in the scope of a transaction must either all be successfully performed or not be performed at all.

JDBC AutoCommit will cause each individual SQL statement as to be executed in its own transaction and the transaction committed when each statement is completed. This makes it impossible to perform operations that consist of multiple SQL statements as a unit of work.
JDBC AutoCommit can be disabled by calling the *setAutoCommit* method with the value false on a JDBC connection.

References:

- Wikipedia: ACID
- Oracle: The Java Tutorials – Using Transactions
- Java 8 API Documentation: Connection

What do you need to do in Spring if you would like to work with JPA?

The following steps are needed if you want to work with JPA in a Spring application:

- Declare the appropriate dependencies.
  In a Maven application, this is accomplished by creating dependencies in the pom.xml file.
  The dependencies in question are typically the ORM framework dependency, a database driver dependency and a transaction manager dependency.
- Implement entity classes with mapping metadata in the form of annotations.
  As a minimum, entity classes need to be annotated with the @Entity annotation on class level and the @Id annotation annotating the field or property that is to be the primary key of the entity.
  It is possible to separate mapping metadata from the implementation of entity classes by using an orm.xml file, but that is outside of the scope of this book.
- Define an *EntityManagerFactory* bean.
  The JPA support in the Spring framework offer three alternatives when creating an *EntityManagerFactoryBean*:
  1. *LocalEntityManagerFactoryBean*
  Use this option, which is the simplest option, in applications that only use JPA for persistence and in integration tests.
  2. Obtain an *EntityManagerFactory* using JNDI
  Use this option if the application is run in a JavaEE server.
  3. *LocalContainerEntityManagerFactoryBean*
  Gives the application full JPA capabilities.
- Define a *DataSource* bean.
- Define a *TransactionManager* bean.
  Typically using the *JpaTransactionManager* class from the Spring Framework.
- Implement repositories.
  A better alternative is using Spring Data JPA, with which you only need to create repository interfaces.

EntityManagerFactory

An entity manager factory is used to interact with a persistence unit. On the entity manager factory the following are typically configured:

- An adapter for the ORM implementation to be used by the application, for example EclipseLink or Hibernate.
- The type of database used by the application.

- ORM configuration properties.
- The package(s) in the application in which to scan for entities.

References:

- JavaEE 7 API: EntityManagerFactory
- JavaSE 8 API: DataSource
- JavaEE 7 API: TransactionManager
- Spring 5 API Documentation: LocalEntityManagerFactoryBean
- Spring 5 API Documentation: JpaTransactionManager
- Spring 5 API Documentation: LocalContainerEntityManagerFactoryBean
- Spring 5 Reference: JPA

Are you able to participate in a given transaction in Spring while working with JPA?

The short answer is: Yes.

The Spring *JpaTransactionManager* supports direct *DataSource* access within one and the same transaction allowing for mixing plain JDBC code that is unaware of JPA with code that use JPA.

If the Spring application is to be deployed to a JavaEE server, then *JtaTransactionManager* can be used in the Spring application. *JtaTransactionManager* will delegate to the JavaEE server's transaction coordinator.

References:

- Spring 5 Reference: Spring-driven JPA transactions
- Spring 5 API Documentation: JpaTransactionManager
- Spring 5 API Documentation: JtaTransactionManager

Which PlatformTransactionManager (s) can you use with JPA?

First, any JTA transaction manager can be used with JPA since JTA transactions are global transactions, that is they can span multiple resources such as databases, queues etc. Thus JPA persistence becomes just another of these resources that can be involved in a transaction.

When using JPA with one single entity manager factory, the Spring Framework *JpaTransactionManager* is the recommended choice. This is also the only transaction manager that is JPA entity manager factory aware.
If the application has multiple JPA entity manager factories that are to be transactional, then a JTA transaction manager is required.

Reference:

- Spring 5 Reference: Understanding the Spring Framework transaction abstraction
- Spring 5 API Documentation: JpaTransactionManager

What do you have to configure to use JPA with Spring? How does Spring Boot make this easier?

What do you have to configure to use JPA with Spring?

Please refer to the earlier section on What do you need to do in Spring if you would like to work with JPA.

How does Spring Boot make this easier?

Spring Boot provides a starter module that:

- Provides a default set of dependencies needed for using JPA in a Spring application.
- Provides all the Spring beans needed to use JPA.
  These beans can be easily customized by declaring bean(s) with the same name(s) in the application, as is standard in Spring applications.
- Provides a number of default properties related to persistence and JPA.
  These properties can be easily customized by declaring one or more properties in the application properties-file supplying new values.

Chapter 4

Spring Data JPA

What is a Repository interface?

A repository interface, also known as a Spring Data repository, is a repository that need no implementation and that supports the basic CRUD (create, read, update and delete) operations. Such a repository is declared as an interface that typically extend the *Repository* interface or an interface extending the *Repository* interface. The *Repository* uses Java generics and takes two type arguments; an entity type and a type of the primary key of entities.

References:

- Spring Data JPA Reference: Core concepts
- Wikipedia: Create, read, update and delete
- Spring Data JPA API Documentation: Repository
- Spring Data Commons Reference: Defining repository interfaces

How do you define a Repository interface? Why is it an interface not a class?

How do you define a Repository interface?

Before being able to define the repository interface there must be a properly annotated entity class. In the example in this section this will be a *Person* entity class.
In addition the example also use a custom primary key class. Such a class need to be annotated with the @Embeddable annotation and the field in the entity class containing the primary key need to be annotated with the @EmbeddedId class. If using a regular primary key class, like *Long*, *Integer* or *String*, then the primary key field in the entity class must be annotated with the @Id annotation.

The following example shows how a repository that store entities of the type *Person*, each with an id of the type *SocialSecurityNumber* is declared:

```
public interface PersonRepository extends JpaRepository<Person, SocialSecurityNumber> {
}
```

Note that the body of the interface is empty. If only basic CRUD operations and basic find-methods are the only types of methods needed in the repository, then no methods need to be declared in the interface.
It is possible to define custom finder methods in the repository interface which Spring Data will implement – more on this in subsequent sections.

With the Spring Data repository interfaces in place annotate a Spring configuration class with an annotation, @EnableJpaRepositories in the case of Spring Data JPA, to enable the discovery and creation of repositories.

Why is it an interface not a class?

The first and most obvious reason for using an interface to define a Spring Data repository is that there usually is no need for implementation of the methods defined in the interface.
Repositories are defined as interfaces in order for Spring Data to be able to use the JDK dynamic proxy mechanism to create the proxy objects that intercept calls to repositories.
This also allows for supplying a custom base repository implementation class that will act as a (custom) base class for all the Spring Data repositories in the application.

References:

- Spring Data JPA Reference: Defining Repository interfaces
- Spring Data JPA API: @EnableJpaRepositories
- Java EE 7 API: @EmbeddedId
- Java EE 7 API: @Embeddable
- Java EE 7 API: @Id
- Spring Data Commons Reference: Working with Spring Data repositories
- Spring Data Commons Reference: Customize the base repository

What is the naming convention for finder methods in a Repository interface?

As earlier, it is possible to add custom finder methods to Spring Data repository interfaces. If following the naming convention below, Spring Data will recognize these find methods and supply an implementation for these methods. The naming convention of these finder methods are:

```
find(First[count])By[property expression][comparison  operator][ordering operator]
```

- Finder method names always start with “find”.
- Optionally, “First” can be added after “find” in order to retrieve only the first found entity. When retrieving only one single entity, the finder method will return null if no matching entity is found. Alternatively the return-type of the method can be declared to use the Java *Optional* wrapper to indicate that a result may be absent.
  If a count is supplied after the “First”, for example “findFirst10”, then the count number of entities first found will be the result.
- The optional property expression selects the property of a managed entity that will be used to select the entity/entities that are to be retrieved.
  Properties may be traversed, in which case underscore can be added to separate names of nested properties to avoid disambiguities. If the property to be examined is a string type, then “IgnoreCase” may be added after the property name in order to perform case-insensitive comparison.
  Multiple property expressions can be chained using “AND” or “OR”.
- The optional comparison operator enables creation of finder methods that selects a range of entities.
  Some comparison operators available are:
  LessThan, GreaterThan, Between, Like.
- Finally the optional ordering operator allows for ordering a list of multiple entities on a property in the entity.
  This is accomplished by adding “OrderBy”, a property expression and “Asc” or “Desc”.
  Example: findPersonByLastnameOrderBySocialsecuritynumberDesc – find persons that have a supplied last name and order them in descending order by social security number.

References:

- Spring Data JPA Reference: Null handing of Repository Methods
- Spring Data JPA Reference: Defining query methods

How are Spring Data repositories implemented by Spring at runtime?

For a Spring Data repository a JDK dynamic proxy is created which intercepts all calls to the repository. The default behavior is to route calls to the default repository implementation, which in Spring Data JPA is the *SimpleJpaRepository* class. It is possible to customize either the implementation of one specific repository type or customize the implementation used for all repositories.
In the former case, customization of one repository type, the proxy for the particular type will invoke any method(s) implemented in the custom implementation, or, if no method exist in the custom implementation, invoke the default method.
In the latter case, customization applied to all repository types, the proxies for repositories will invoke any method(s) implemented in the custom implementation, or, if no method exist in the custom implementation, invoke the default method.

References:

- Stack Overflow: How are Spring Data repositories actually implemented?
- Spring Data JPA API: SimpleJpaRepository
- Spring Data JPA Reference: Custom Implementations for Spring Data Repositories

What is @Query used for?

The @Query annotation allows for specifying a query to be used with a Spring Data JPA repository method. This allows for customizing the query used for the annotated repository method or supplying a query that is to be used for a repository method that do not adhere to the finder method naming convention described earlier.

Example:

```
public interface PersonRepository extends JpaRepository<Person, Long> {
  @Query("select p from Person p where p.emailAddress = ?1")
  Person findByEmailAddress(String emailAddress);
}
```

References:

- Spring Data JPA API: @Query
- Spring Data JPA Reference: Using @Query

Chapter 5

Spring MVC and the Web Layer

MVC is an abbreviation for a design pattern. What does it stand for and what is the idea behind it?

MVC is an abbreviation for Model-View-Controller, which is a design pattern used to separate concerns of an application that has a user interface, such as web applications.
As the name suggests, there are three main parts in this pattern:

- Model
  The model holds the current data and business logic of the application.
- View
  The view is responsible for presenting the data of the application to the user. The user interacts with the view.
- Controller
  Controllers acts as a mediator between the model and the view accepting requests from the view, issuing commands to the model to manipulate the data of the application and finally interacts with the view to render the result.

Some advantages expected from using MVC are:

- Reuse of model and controllers with different views.
  Example: One view for presentation of the application as a web application and another view for mobile devices, both using the same model and controllers.
- Reduced coupling between the model, view and controller.
  This is compared to if implementing an application not using MVC.
- Separation of concerns.
  This in turn result in increased maintainability and extensibility. It also allows for more easily sharing the work of developing the application between multiple developers.

References:

- Wikipedia: Model-View-Controller
- Spring 5 Reference Documentation: Spring Web MVC

What is the DispatcherServlet and what is it used for?

The *DispatcherServlet* is a servlet that implement the front controller design pattern. The duties of the *DispatcherServlet* consist of:

- Receives requests and delegates them to registered handlers.
  If only one *DispatcherServlet* is used in the application, then this servlet will receive all requests that are sent to the application.
- Resolves views by mapping view-names to *View* instances.
- Resolves exceptions that occur during handler mapping or execution.
  The most common is to map an exception to an error view.

The front controller design pattern allows for centralizing matters, like security and error handling, that are to be applied to the entire application.

A Spring web application may define multiple dispatcher servlets, each of which has its own namespace, its own Spring application context and its own set of mappings and handlers.

References:

- Spring 5 Reference: Spring Web MVC – DispatcherServlet
- Wikipedia: Front Controller
- Martin Fowler: P of EAA Catalog – Front Controller
- Spring 5 API Documentation: DispatcherServlet
- Spring 5 API Documentation: ViewResolver
- Spring 5 API Documentation: View
- Spring 5 API Documentation: HandlerExceptionResolver

What is a web application context? What extra scopes does it offer?

Web application context, specified by the *WebApplicationContext* interface, is a Spring application context for a web applications. It has all the properties of a regular Spring application context, given that the *WebApplicationContext* interface extends the *ApplicationContext* interface, and add a method for retrieving the standard Servlet API *ServletContext* for the web application.

In addition to the standard Spring bean scopes singleton and prototype, there are three additional scopes available in a web application context:

Scope	Description
request	Single bean instance per HTTP request.
session	Single bean instance per HTTP session.
application	Single bean instance per *ServletContext*.

References:

- Spring 5 Reference: Spring Web MVC – Context Hierarchy
- Spring 5 API Documentation: WebApplicationContext

What is the @Controller annotation used for?

The @Controller annotation is a specialization of the @Component annotation that have been discussed earlier. It is used to annotate classes that implement web controllers, the C in MVC. Such classes need not extend any particular parent class or implement any particular interfaces.

References:

- Spring 5 API Documentation: @Controller
- Spring 5 Reference: Spring Web MVC – Annotated Controllers

How is an incoming request mapped to a controller and mapped to a method?

To enable requests to be mapped to one or more methods in a controller class the following steps are taken:

- Enable component scanning.
  This enables auto-detection of classes annotated with the @Controller annotation.
  In Spring Boot applications, the @SpringBootApplication annotation is an annotation that itself is annotated with a number of meta-annotation one of which is the @ComponentScan annotation.
- Annotate one of the configuration-classes in the application with the @EnableWebMvc annotation.
  In Spring Boot applications, it is sufficient to have one configuration-class in the application implement the *WebMvcConfigurer* interface.
- Implement a controller class that is annotated with the @Controller annotation.
  Controller classes can also be annotated with the @RequestMapping annotation, in which case it will add a part to the URL which will map to controller method(s).
  Example: Controller class is annotated with @RequestMapping("/c2") and a method in the controller class is annotate with @RequestMapping("/greeting"). A request to http://localhost:8080/c2/greeting will be mapped to the to the method in the controller class in the example.
- Implement at least one method in the controller class and annotate the method with @RequestMapping.
  Instead of the @RequestMapping annotation, one of the specialized annotations can be used. An example is @GetMapping.
  This type of methods are called controller method or handler method.

When a request is issued to the application:

- The/a *DispatcherServlet* of the application receives the request.
- The/a *DispatcherServlet* maps the request to a method in a controller.
  The *DispatcherServlet* holds a list of classes implementing the *HandlerMapping* interface.
- The/a *DispatcherServlet* dispatches the request to the controller.
- The method in the controller is executed.

References:

- Spring 5 Reference: Spring Web MVC – Request Mapping
- Spring 5 Reference: Spring Web MVC – Handler Methods
- Spring 5 API Documentation: @Controller

- Spring 5 API Documentation: @RequestMapping
- Spring 5 API Documentation: @GetMapping
- Spring 5 API Documentation: @PostMapping
- Spring 5 API Documentation: @PutMapping
- Spring 5 API Documentation: @DeleteMapping
- Spring 5 API Documentation: @PatchMapping
- Spring 5 API Documentation: WebMvcConfigurer
- Spring 5 API Documentation: DispatcherServlet
- Spring 5 API Documentation: HandlerMapping

What is the difference between @RequestMapping and @GetMapping?

The @RequestMapping annotation will cause requests using the HTTP method(s) specified in the optional *method* element of the annotation to be mapped to the annotated controller method.
If no *method* element specified in the @RequestMapping annotation, then requests using any HTTP method will be mapped to the annotated method.

The @GetMapping annotation is a specialization of the @RequestMapping annotation with *method* = RequestMethod.GET. Thus only requests using the HTTP GET method will be mapped to the method annotated by @GetMapping.

References:

- Spring 5 Reference: Spring Web MVC – Request Mapping
- Spring 5 API Documentation: @RequestMapping
- Spring 5 API Documentation: @GetMapping

What is @RequestParam used for?

The @RequestParam annotation is used to annotate parameters to handler methods in order to bind request parameters to method parameters.
Assume there is a controller method with the following signature and annotations:

```
    @RequestMapping("/greeting")
    public String greeting(@RequestParam(name="name", required=false) String inName) {
        …
    }
```

If then a request is sent to the URL http://localhost:8080/greeting?name=Ivan then the *inName* method parameter will contain the string “Ivan”.
The first request parameter in an URL is preceded with a question mark. Subsequent request parameters are preceded with an ampersand.
Example:

```
http://localhost:8080/greeting?firstName=Ivan&lastName=Krizsan
```

References:

- Spring 5 API Documentation: @RequestParam
- Spring 5 Reference: Spring Web MVC – Handler Methods
- Spring 5 Reference: Spring Web MVC - @RequestParam

What are the differences between @RequestParam and @PathVariable?

Request Parameters

As in the previous section, the following example shows an URL with two request parameters:

```
http://localhost:8080/greeting?firstName=Ivan&lastName=Krizsan
```

The first request parameters has the name firstName and the value Ivan, the second request parameter has the name lastName and the value Krizsan.

Path Variables

The following shows an URL from which one could extract two path variables:

```
http://localhost:8080/firstname/Ivan/lastname/Krizsan
```

The following handler method signature is a handler method that will map the “Ivan” part of the above URL to a method parameter named *inFirstName* and the “Krizsan” part of the above URL to another method parameter named *inLastName*:

```
    @RequestMapping("/firstname/{firstName}/lastname/{lastName}")
    public String greetWithFirstAndLastName(
        @PathVariable("firstName") final String inFirstName,
        @PathVariable("lastName") final String inLastName) {
```

Note that:

- The part of the URL in the @RequestMapping annotation contains template variables.
  In the above examples these are firstName and lastName and they are surrounded by curly brackets.
- The values in the @PathVariable annotations match the name of the template variables in the value of the @RequestMapping annotation.
  This is not necessary if the method parameters have the same names as the template variables.
- In the Spring literature firstName and lastName are also called URI template variables.

Difference

The difference between the @RequestParam annotation and the @PathVariable annotation is that they map different parts of request URLs to handler method arguments.
@RequestParam maps query string parameters to handler method arguments.
@PathVariable maps a part of the URL to handler method arguments.

References:

- Spring 5 API Documentation: @RequestParam
- Spring 5 API Documentation: @PathVariable

- Spring 5 Reference: Spring MVC – Request Mapping
- Wikipedia: Query String

What are some of the parameter types for a controller method?

The following table lists the different types of arguments that can be used in controller methods. Controller method arguments of a type not included in the table below will as default be handled as if annotated with @RequestParam if the type is a simple type or handled as if annotated with @ModelAttribute otherwise.

Controller Method Argument Type	Gives Access To
WebRequest	Request metadata such as context path, request parameters, user principal etc.
NativeWebRequest	Extension of WebRequest that also allows access to native request and response objects.
javax.servlet.ServletRequest	Request object. The Spring MultipartRequest type can also be used.
javax.servlet.ServletResponse	Response object.
javax.servlet.http.HttpSession	HTTP session object. Will ensure that a session object exists and will create one if this is not the case.
javax.servlet.http.PushBuilder	Servlet 4.0 builder used to build push requests.
java.security.Principal	Currently authenticated user principal.
HttpMethod	Request HTTP method.
java.util.Locale	Locale of the current request.
java.util.TimeZone, java.time.ZoneId	Time zone associated with the current request.
java.io.InputStream, java.io.Reader	Request body.
java.io.OutputStream, java.io.Writer	Response body.
HttpEntity<B>	Request headers and body.
java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap	Model that is used in controllers and exposed when the view is rendered.
RedirectAttributes	Extends the Model interface allowing selection of attributes for redirect scenarios.
Errors	Data-binding and validation errors.
BindingResult	Extends the Errors interface to allow registration of errors, application of a Validator and binding-specific analysis and model building.
SessionStatus	Session processing status; getting and setting.
UriComponentsBuilder	Builder for creating URI references. Supports URI templates.

References:

- Spring 5 Reference: Spring MVC - Handler Methods – Method Arguments
- Spring 5 API Documentation: WebRequest
- Spring 5 API Documentation: NativeWebRequest
- Spring 5 API Documentation: HttpMethod
- Spring 5 API Documentation: HttpEntity
- Spring 5 API Documentation: RedirectAttributes
- Spring 5 API Documentation: Errors
- Spring 5 API Documentation: BindingResult
- Spring 5 API Documentation: SessionStatus
- Spring 5 API Documentation: UriComponentsBuilder

What other annotations might you use on a controller method parameter? (You can ignore form-handling annotations for the exam)

The following table lists annotations that can be applied to arguments of controller methods.

Controller Method Argument Annotation	Function
@PathVariable	Binds a part of the URL, a path segment, to a handler method argument.
@MatrixVariable	Binds a name-value pair in a part of the URL, a path segment, to a handler method argument.
@RequestParam	Binds a query string parameter to a handler method arguments.
@RequestHeader	Binds a request header to a handler method argument.
@CookieValue	Binds the value of a HTTP cookie to a handler method argument.
@RequestBody	Binds the body of a request to a handler method argument.
@RequestPart	Binds a part of a "multipart/form-data" request to a handler method argument.
@ModelAttribute	Binds a named model attribute to a handler method argument or the return value of a handler method.
@SessionAttributes	Causes selected model attributes to be stored in the HTTP Servlet session between requests.

@SessionAttribute	Binds a session attribute to a handler method argument.
@RequestAttribute	Binds a request attribute to a handler method argument.

References:

- Spring 5 Reference: Spring MVC - Handler Methods – Method Arguments
- Spring 5 API Documentation: @PathVariable
- Spring 5 API Documentation: @MatrixVariable
- Spring 5 Reference: Spring MVC – Matrix variables
- Spring 5 API Documentation: @RequestParam
- Spring 5 API Documentation: @RequestHeader
- Spring 5 Reference: Spring MVC – Handler methods - @RequestHeader
- Spring 5 API Documentation: @CookieValue
- Spring 5 Reference: Spring MVC – Handler methods - @CookieValue
- Spring 5 API Documentation: @RequestBody
- Spring 5 Reference: Spring MVC – Handler methods - @RequestBody
- Spring 5 API Documentation: @RequestPart
- Spring 5 Reference: Spring MVC – Handler methods - Multipart
- Spring 5 API Documentation: @ModelAttribute
- Spring 5 Reference: Spring MVC – Handler methods - @ModelAttribute
- Spring 5 API Documentation: @SessionAttributes
- Spring 5 Reference: Spring MVC – Handler methods - @SessionAttributes
- Spring 5 API Documentation: @SessionAttribute
- Spring 5 Reference: Spring MVC – Handler methods - @SessionAttribute
- Spring 5 API Documentation: @RequestAttribute
- Spring 5 Reference: Spring MVC – Handler methods - @RequestAttribute

What are some of the valid return types of a controller method?

The following table lists valid return types of controller methods:

Controller Method Return Type	Description
HttpEntity<B>	As a return type from a controller method, represents a response entity consisting of headers and a body.
ResponseEntity<B>	Extension of HttpEntity that adds a HTTP status code.
HttpHeaders	Used when response only consists of HTTP headers and no body.
String	Name of view to be rendered.
View	An object that, given a model, a request and a response, renders a view.
java.util.Map org.springframework.ui.Model	Map or Model instance holding attributes to be added to the model.
ModelAndView	Object holding both model and view.
void	Used when controller method handles responses by writing to an output stream or if the method is annotated with @ResponseStatus. Used in REST controller methods that are not to return a response body. Used in HTML controller methods selecting the default view name. Returning null from a controller method produces the same result.
DeferredResult<V> alternatively ListenableFuture<V>, java.util.concurrent.CompletionStage<V>, java.util.concurrent.CompletableFuture<V>	Asynchronously produce the return value of the controller method from another thread.
Callable<V>	Asynchronously produce the return value of the controller method from a Spring MVC managed thread.
ResponseBodyEmitter	Asynchronously process requests allowing objects to be written to the response.
SseEmitter	Specialization of ResponseBodyEmitter that allows for sending Server-Sent Events.
StreamingResponseBody	Asynchronously process requests allowing the application to write to the response output

	stream, bypassing message conversion, without blocking the servlet container thread.
Reactive types	Alternative to DeferredResult for use with Reactor, RxJava or similar.
Any other return value type	By default treated as a view name. Simple types are returned unresolved.

The following table lists annotations which can be applied to controller methods that affect the result returned:

Controller Method Annotation	Description
@ResponseBody	Response is created from the serialized result of the controller method result processed by a HttpMessageConverter.
@ModelAttribute	Model attribute with name specified in the annotation is added to the model with value being the result of the controller method.

Refereences:

- Spring 5 Reference: Spring MVC – Handler methods – Return Values
- Spring 5 API Documentation: HttpEntity
- Spring 5 API Documentation: ResponseEntity
- Spring 5 Reference: Spring MVC - ResponseEntity
- Spring 5 API Documentation: HttpHeaders
- Spring 5 API Documentation: View
- Spring 5 API Documentation: Model
- Spring 5 API Documentation: ModelAndView
- Spring 5 API Documentation: DeferredResult
- Spring 5 Reference: Async Requests
- Spring 5 API Documentation: ListenableFuture
- Spring 5 API Documentation: ResponseBodyEmitter
- Spring 5 API Documentation: SseEmitter
- Spring 5 API Documentation: StreamingResponseBody
- Spring 5 API Documentation: @ResponseBody

- Spring 5 API Documentation: @ModelAttribute
- Spring 5 Reference: Spring MVC – Controllers
- Spring 5 Reference: Spring MVC – Message Conversion

Chapter 6

Security

What are authentication and authorization? Which must come first?

Authentication and authorization are two fundamental concepts in the realm of application security.

Authentication

The short explanation of authentication is that it is the process of verifying that, for instance, a user of a computer system is who he/she claims to be.

Authentication is the process of confirming that some piece of information is true. An example is the login of a computer system; a user enters a user name and a password. The password is a secret that is used to confirm that the person entering the username is indeed the user he/she claims to be. The user proves his/her identity to the computer system.

Apart from the traditional username and password way of authentication, there are other ways of authenticating a user of a computer system such as PIN numbers, security questions, id cards, fingerprints etc etc.

In Spring Security, the authentication process consists of the following steps quoted from the Spring Security reference:

- The username and password are obtained and combined into an instance of *UsernamePasswordAuthenticationToken* (an instance of the *Authentication* interface).
- The token is passed to an instance of *AuthenticationManager* for validation.
- The *AuthenticationManager* returns a fully populated *Authentication* instance on successful authentication.
- The security context is established by calling *SecurityContextHolder.getContext().setAuthentication(…)*, passing in the returned authentication object.

Authorization

The short explanation of authorization is the process of determining that a user of a computer system is permitted to do something that the user is attempting to do.

Authorization is the process of specifying access rights to resources. For instance, the only type of users that can create and delete users in a computer system is users is users in the administrator role. Thus the only users that have access to the create and delete functions of the application are users in the administrator role.

Which must come first?

Unless there is some type of authorization that specifies what resources and/or functions that can be accessed by anonymous users, authentication must always come before authorization.

This in order to be able to establish the identity of the user before being able to verify what the user is allowed to do.

References:

- Wikipedia: Authentication
- Wikipedia: Authorization
- Spring Security 5 Reference: Authentication

Is security a cross cutting concern? How is it implemented internally?

Security – A Cross-Cutting Concern?

Short answer: Yes, security is a cross cutting concern.

Security is a function of an application that is not immediately associated with the business logic of the application – it is a secondary or supporting function. In addition, security may be required for many different areas of functionality of an application and thus does not fit well in software development primarily concerned with decomposition (of an application) by functionality.

Quoted from the Wikipedia article on aspect-oriented software development:

“Aspect-oriented software development focuses on the identification, specification and representation of cross-cutting concerns and their modularization into separate functional units”

The Wikipedia article on Cross-Cutting Concern lists “information security” as an example of a concern that tend to be cross-cutting.

Finally, the Spring Security FAQ states that:

“Authorization is a crosscutting concern”

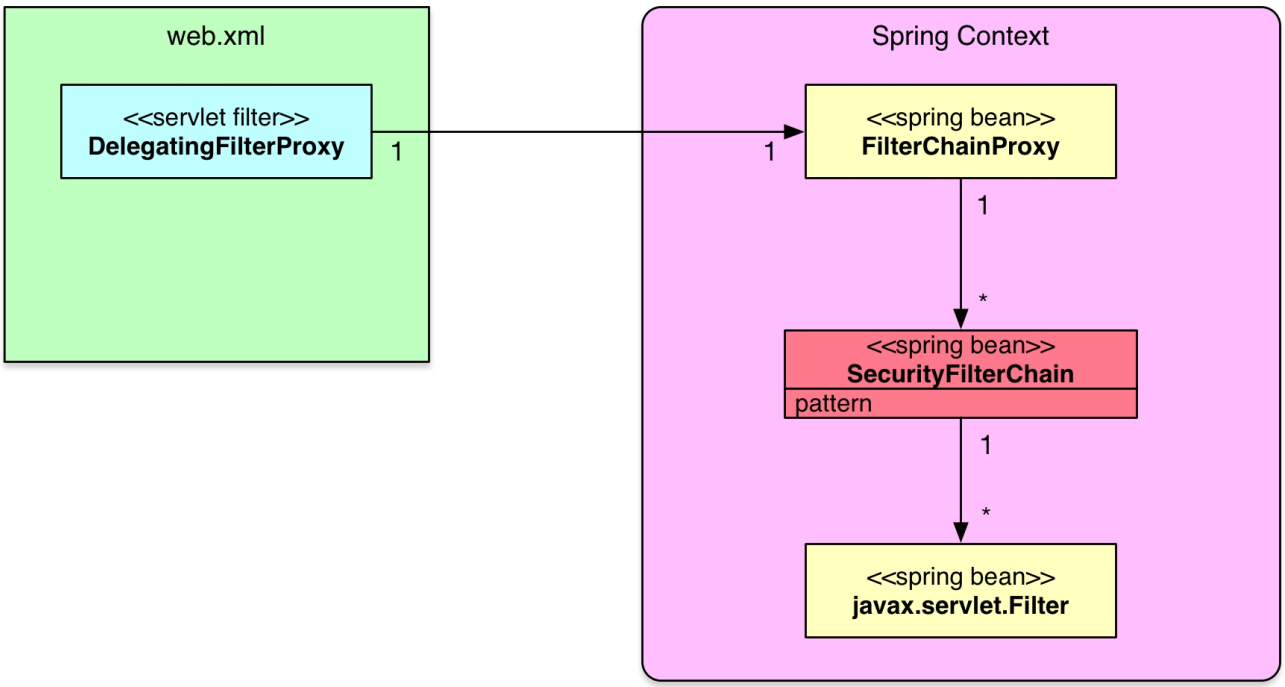
How is security implemented internally in Spring Security?

Spring Security is implemented in the following two ways depending on what is to be secured:

- Using a Spring AOP proxy that inherits from the *AbstractSecurityInterceptor* class. Applied to method invocation authorization on objects secured with Spring Security. Please refer to the section on Aspect Oriented Programming for details on Spring AOP proxies.
- Spring Security’s web infrastructure is based entirely on servlet filters. Details below.

Spring Security Web Infrastructure

The following figure shows an overview of the Spring Security web infrastructure:



Spring Security web infrastructure overview.

First of all a servlet filter of the type *DelegatingFilterProxy* is configured. For details on the *DelegatingFilterProxy*, please see separate section below.

The *DelegatingFilterProxy* delegates to a *FilterChainProxy*. The *FilterChainProxy* is defined as a Spring bean and takes one or more *SecurityFilterChain* instances as constructor parameter(s).

A *SecurityFilterChain* associates a request URL pattern with a list of (security) filters. For details on the *SecurityFilterChain*, please see separate section below.

An example configuration is shown below:

```
    @Bean
    public FilterChainProxy myFilterChainProxy(
        @Qualifier("securityContextPersistenceFilterWithASCFalse")
        final SecurityContextPersistenceFilter
            inSecurityContextPersistenceFilterWithASCFalse,
        @Qualifier("securityContextPersistenceFilterWithASCTrue")
        final SecurityContextPersistenceFilter
            inSecurityContextPersistenceFilterWithASCTrue,
        final BasicAuthenticationFilter inBasicAuthenticationFilter,
        final UsernamePasswordAuthenticationFilter inFormLoginFilter,
        final ExceptionTranslationFilter inExceptionTranslationFilter,
        final FilterSecurityInterceptor inFilterSecurityInterceptor) {

        /* Create the filter chain for the RESTful services. */
        final MvcRequestMatcher theRestfulServicesRequestMatcher =
            new MvcRequestMatcher(
```

```
                new HandlerMappingIntrospector(),
                "/restful/**");

        final DefaultSecurityFilterChain theRestfulServicesSecurityFilterChain =
            new DefaultSecurityFilterChain(
                theRestfulServicesRequestMatcher,
                /* Filters start here. */
                inSecurityContextPersistenceFilterWithASCFalse,
                inBasicAuthenticationFilter,
                inExceptionTranslationFilter,
                inFilterSecurityInterceptor);

        /* Create the default filter chain for all resources of the application. */
        final MvcRequestMatcher theAllUrlsRequestMatcher = new MvcRequestMatcher(
            new HandlerMappingIntrospector(),
            "/**");
        final DefaultSecurityFilterChain theDefaultSecurityFilterChain =
            new DefaultSecurityFilterChain(
                theAllUrlsRequestMatcher,
                /* Filters start here. */
                inSecurityContextPersistenceFilterWithASCTrue,
                inFormLoginFilter,
                inExceptionTranslationFilter,
                inFilterSecurityInterceptor);

        /*
         * Create the filter chain proxy adding the two security filter chains
         * created above. Note that the order in which the security filter
         * chains are added is significant:
         * The security filter chain for the RESTful services must be before the
         * default security filter chain, otherwise it will never be matched since
         * the default security filter chain will match all requests and the first
         * matching security filter chain will be used for each request.
         */
        return new FilterChainProxy(
            Arrays.asList(
                theRestfulServicesSecurityFilterChain,
                theDefaultSecurityFilterChain)
        );
    }
```

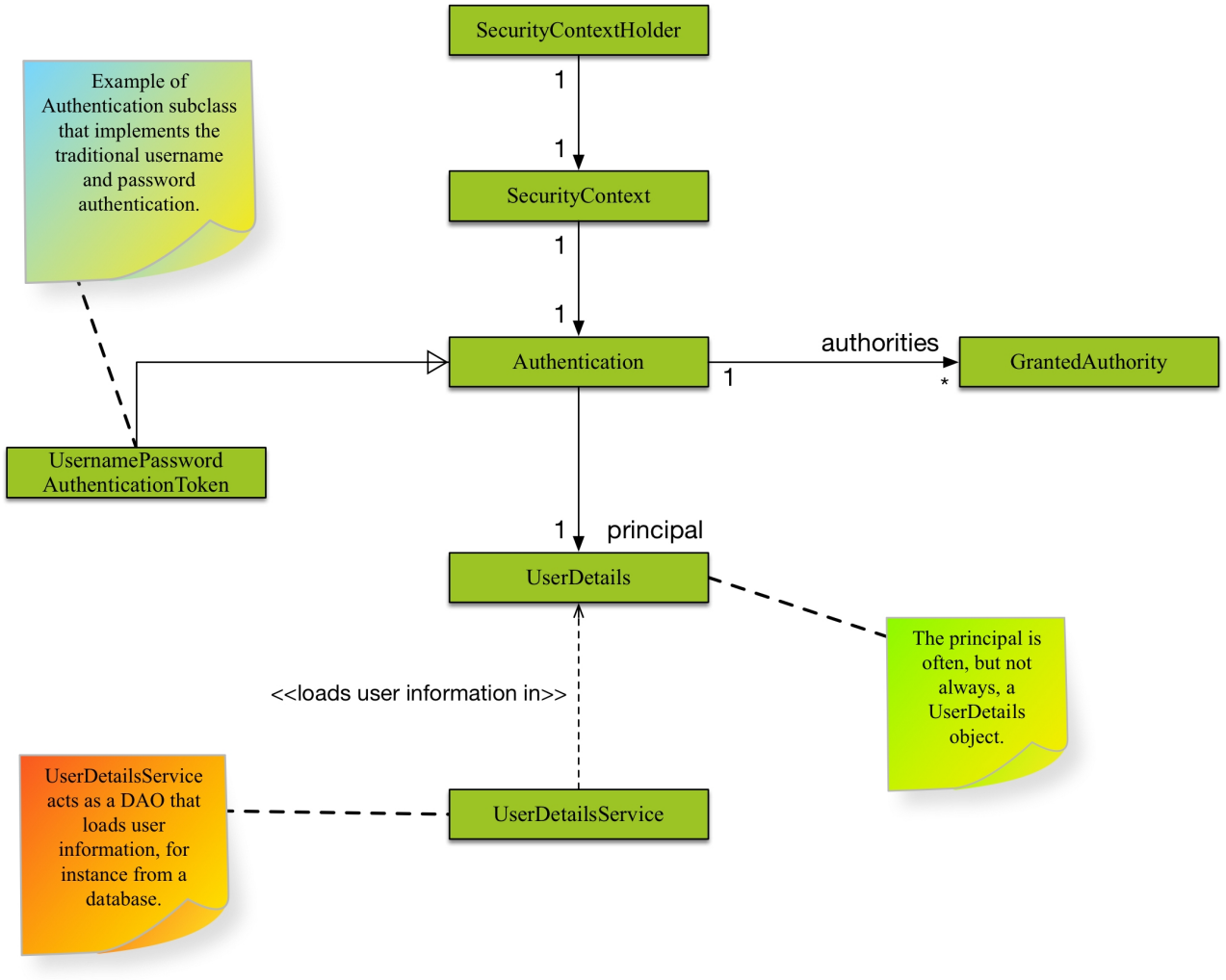
The above is the example from FilterChainProxy section in the Spring Security reference re-written using JavaConfig.

Spring Security Core Components

Core components in Spring Security are:

Component Type	Function
SecurityContextHolder	Contains and provides access to the *SecurityContext* of the application. Default behavior is to associate the *SecurityContext* with the current thread.
SecurityContext	Default and only implementation in Spring Security holds an *Authentication* object. May also hold additional request-specific information.
Authentication	Represents token for authentication request or authenticated principal after the request has been granted. Also contains the authorities in the application that an authenticated principal has been granted.
GrantedAuthority	Represents an authority granted to an authenticated principal.
UserDetails	Holds user information, such as user-name, password and authorities of the user. This information is used to create an *Authentication* object on successful authentication. May be extended to contain application-specific user information.
UserDetailsService	Given a user-name this service retrieves information about the user in a *UserDetails* object. Depending on the implementation of the user details service used, the information may be stored in a database, in memory or elsewhere if a custom implementation is used.

The relationships between the Spring Security core components are shown in the following figure:



Spring Security 5 core components and their relationships.

References:

- Spring Security 5 Reference: Spring Security FAQ
- Wikipedia: Cross-Cutting Concern
- Wikipedia: Aspect-Oriented Software Development
- Spring Security 5 Reference: Architecture and Implementation – Core Components
- Spring Security 5 Reference: Access-Control (Authorization) in Spring Security
- Spring Security 5 Reference: Web Application Security

What is the delegating filter proxy?

The *DelegatingFilterProxy* class implements the *javax.servlet.Filter* interface and thus is a servlet filter. One delegating filter proxy delegates to one Spring bean that is required to implement the *javax.servlet.Filter* interface. This mechanism allows for defining servlet filters in the web.xml (for the Servlet 2 standard) which later looks up a named bean from the Spring application context and delegates filtering to the Spring bean.

The Spring bean to which a delegating filter proxy delegates to will, as any Spring bean, have its lifecycle handled by the Spring container. The servlet filter lifecycle methods *init* and *destroy* will by default not be called. The delegating filter proxy can be configured to invoke these methods on the Spring bean when the corresponding method is called on the delegating filter proxy by setting the *targetFilterLifecycle* property to true.

References:

- Spring Security 5 API Documentation: DelegatingFilterProxy

What is the security filter chain?

As earlier, a *SecurityFilterChain* associates a request URL pattern with a list of (security) filters.

Example of Java configuration of a security filter chain:

```
    @Bean
    public SecurityFilterChain appRsrcsSecurityFilterChain(
        final FilterSecurityInterceptor inFilterSecurityInterceptor,
        final ExceptionTranslationFilter inExceptionTranslationFilter,
        final SecurityContextPersistenceFilter inSecurityContextPersistenceFilter,
        final ConcurrentSessionFilter inConcurrentSessionFilter,
        final BasicAuthenticationFilter inBasicAuthenticationFilter,
        final SessionManagementFilter inSessionManagementFilter,
        final SecurityContextHolderAwareRequestFilter
            inSecurityContextHolderAwareRequestFilter,
        final LogoutFilter inLogoutFilter) {
        final MvcRequestMatcher theDefaultRequestMatcher = new MvcRequestMatcher(
            new HandlerMappingIntrospector(),
            "/**");


        final DefaultSecurityFilterChain theAppRsrcsSecurityFilterChain =
            new DefaultSecurityFilterChain(
                theDefaultRequestMatcher,
                inSecurityContextPersistenceFilter,
                inConcurrentSessionFilter,
                inLogoutFilter,
                inBasicAuthenticationFilter,
                inSecurityContextHolderAwareRequestFilter,
```

```
                inSessionManagementFilter,
                inExceptionTranslationFilter,
                inFilterSecurityInterceptor);

        return theAppRsrcsSecurityFilterChain;
    }
```

The security filter chain implements the *SecurityFilterChain* interface and the only implementation provided by Spring Security is the *DefaultSecurityFilterChain* class.

There are two parts to a security filter chain; the request matcher and the filters. The request matcher determines whether the filters in the chain are to be applied to a request or not.

The order in which security filter chains are declared is significant, since the first filter chain which has a request URL pattern which matches the current request will be used. Thus a security filter chain with a more specific URL pattern should be declared before a security filter chain with a more general URL pattern.

Request Matcher

There are a number of different request matchers which all implement the *RequestMatcher* interface with perhaps the two most common ones being *MvcRequestMatcher* and *AntPathRequestMatcher.* The former is being used in the above example. The *MvcRequestMatcher* in the example above is configured with the URL pattern "/**", which will match requests to the application with any URL. For example, "http://localhost:8080/myapp/index.html" will be matched and so will "http://localhost:8080/myapp/services/userservice/", assuming the root application URL is "http://localhost:8080/myapp".

Filters

The constructor of the *DefaultSecurityFilterChain* class takes a variable number of parameters, the first always being a request matcher. The remaining parameters are all filters which implements the *javax.servlet.Filter* interface.
The order of the filters in a security filter chain is important – filters must be declared in the following order (filters may be omitted if not needed):

- *ChannelProcessingFilter*
- *SecurityContextPersistenceFilter*
- *ConcurrentSessionFilter*
- Any authentication filter.
  Such as *UsernamePasswordAuthenticationFilter*, *CasAuthenticationFilter*, *BasicAuthenticationFilter*.
- *SecurityContextHolderAwareRequestFilter*
- *JaasApiIntegrationFilter*

- *RememberMeAuthenticationFilter*
- *AnonymousAuthenticationFilter*
- *ExceptionTranslationFilter*
- *FilterSecurityInterceptor*

For further details, please refer to the section on filter ordering in the Spring Security reference.

References:

- Spring Security 5 Reference: The Security Filter Chain
- Spring Security 5 API Documentation: RequestMatcher
- Spring Security 5 API Documentation: AntPathRequestMatcher
- Spring Security 5 API Documentation: SecurityFilterChain
- Spring Security 5 API Documentation: DefaultSecurityFilterChain

What is a security context?

Taking a step back, an object implementing the *SecurityContext* interface is stored in an instance of the *SecurityContextHolder*. The *SecurityContextHolder* class not only keeps a reference to a security context, but it also allows for specifying the strategy used to store the security context:

- Thread local
  A security context is stored in a thread-local variable and available only to one single thread of execution.
- Inheritable thread local
  As thread local, but with the addition that child threads created by a thread containing a thread-local variable containing a reference to a security context will also have a thread-local variable containing a reference to the same security context.
- Global
  A security context is available throughout the application, from any thread.

Taking a look at the *SecurityContext* interface, which defines the minimum security information associated with a thread of execution, there are two methods; one for setting and one for retrieving an object that implements the *Authentication* interface.

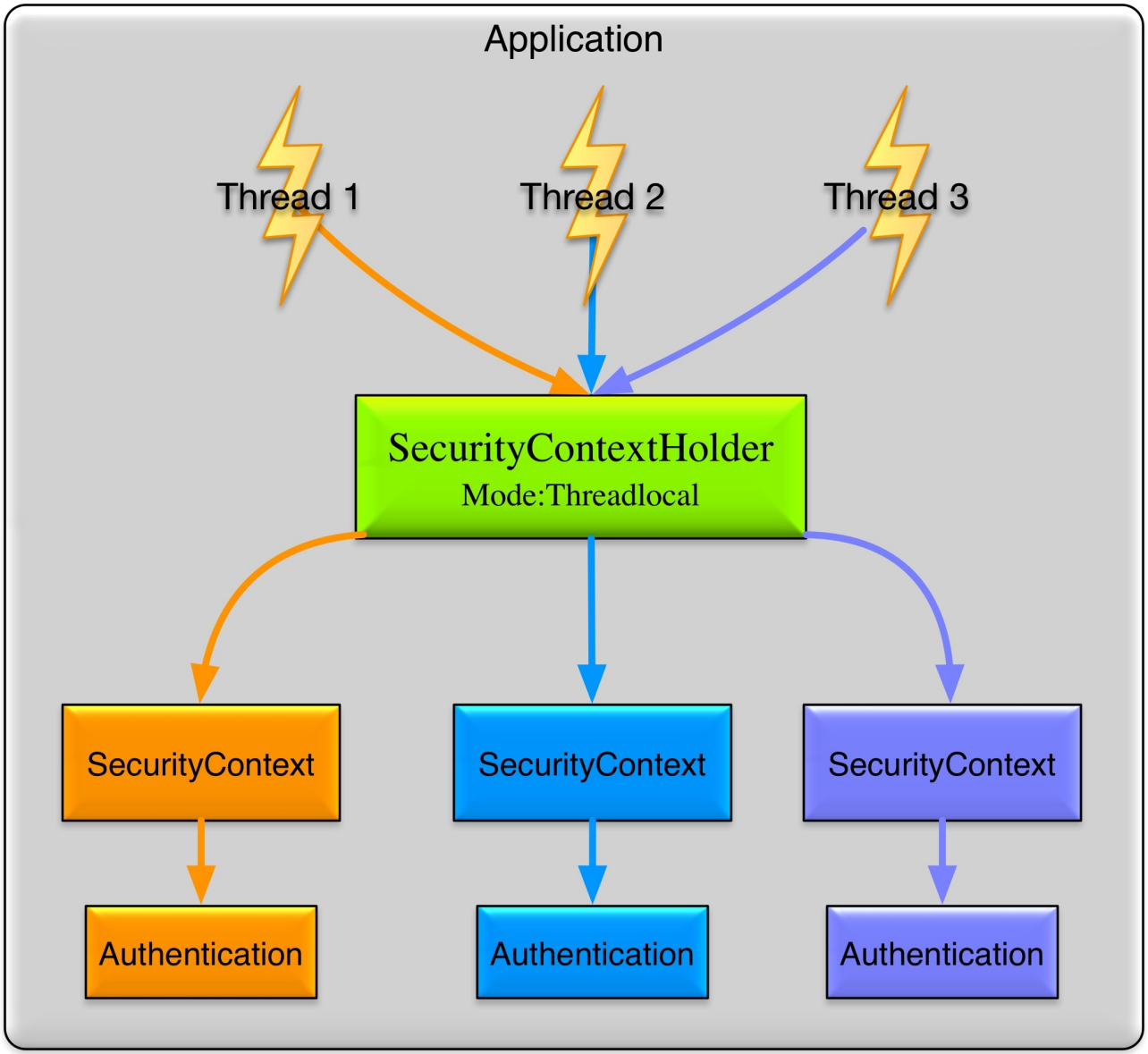
The *Authentication* interface defines the properties of an object that represents a security token for:

- An authentication request
  This is the case prior to a user having been authenticated, when a user tries to log in.
- An authenticated principal
  After a user has been authenticated by an authentication manager.

The basic properties contained in an object implementing the *Authentication* interface are:

- A collection of the authorities granted to the principal.
- The credentials used to authenticate a user.
  This can be a login name and a password that has been verified to match.
- Details
  Additional information, may be application specific or null if not used.
- Principal
- Authenticated flag
  A boolean indicating whether the principal has been successfully authenticated.

There are a number of different implementations of the *Authentication* interface that can be used with different authentication schemes – please refer to the API documentation for further details.



Relationships between SecurityContextHolder and SecurityContexts with threadlocal mode in an application with three different threads, each having its own security context.

References:

- Spring Security 5 API Documentation: SecurityContextHolder
- Spring Security 5 API Documentation: SecurityContext
- Spring Security 5 API Documentation: Authentication
- Spring Security 5 Reference: SecurityContextHolder, SecurityContext and Authentication Objects
- Java SE 8 API Documentation: ThreadLocal
- Java SE 8 API Documentation: InheritableThreadLocal

What does the ** pattern in an antMatcher or mvcMatcher do?

There are two wildcards that can be used in URL patterns:

- *
  Matches any path on the level at which the wildcard occur.
  Example: /services/* matches /services/users and /services/orders but not /services/orders/123/items.
- **
  Matches any path on the level at the wildcard occurs and all levels below.
  If only /** or ** then will match any request.
  Example: /services/** matches /services, /services/, /services/users and /services/orders and also /services/orders/123/items etc.

References:

- Spring Security 5 API: AntPathRequestMatcher

Why is an mvcMatcher more secure than an antMatcher?

As an example *antMatchers("/services")* only matches the exact “/services” URL while *mvcMatchers("/services")* matches “/services” but also “/services/”, “/services.html” and “/services.abc”. Thus the mvcMatcher matches more than the antMatcher and is more forgiving as far as configuration mistakes are concerned. In addition, the *mvcMatchers* API uses the same matching rules as used by the @RequestMapping annotation. Finally, the *mvcMatchers* API is newer than the *antMatchers* API.

Does Spring Security support password hashing? What is salting?

Password Hashing

Password hashing is the process of calculating a hash-value for a password. The hash-value is stored, for instance in a database, instead of storing the password itself. Later when a user attempts to log in, a hash-value is calculated for the password supplied by the user and compared to the stored hash-value. If the hash-values does not match, the user has not supplied the correct password.

In Spring Security, this process is referred to as password encoding and is implemented using the *PasswordEncoder* interface.

Salting

A salt used when calculating the hash-value for a password is a sequence of random bytes that are used in combination with the cleartext password to calculate a hash-value. The salt is stored in cleartext alongside the password hash-value and can later be used when calculating hash-values for user-supplied passwords at login.

The reason for salting is to avoid always having the same hash-value for a certain word, which would make it easier to guess passwords using a dictionary of hash-values and their corresponding passwords.

References:

- Spring Security 5 Reference: Password Encoding
- Spring Security 5 API: PasswordEncoder

Why do you need method security? What type of object is typically secured at the method level (think of its purpose not its Java type).

So far mainly the part of Spring Security that supplies security for web resources, accomplished using servlet filters, has been discussed. Spring Security also has support for security on method level with which security constraints can be applied to individual methods in Spring beans.

Security on the method level needs to be explicitly enabled using the @EnableGlobalMethodSecurity annotation in regular Spring applications or the @EnableReactiveMethodSecurity annotation in reactive Spring applications.

Why do you need method security?

Method security is an additional level of security in web applications but can also be the only layer of security in applications that do not expose a web interface.

What type of object is typically secured at method level?

Method-level security is commonly applied to services in the service layer of an application.

References:

- Spring Security 5 API: @EnableGlobalMethodSecurity

What do @PreAuthorized and @RolesAllowed do? What is the difference between them?

The @PreAuthorize and @RolesAllowed annotations are annotations with which method security can be configured either on individual methods or on class level. In the latter case the security constraints will be applied to all methods in the class.

@PreAuthorize

The @PreAuthorize annotation allows for specifying access constraints to a method using the Spring Expression Language (SpEL). These constraints are evaluated prior to the method being executed and may result in execution of the method being denied if the constraints are not fulfilled. The @PreAuthorize annotation is part of the Spring Security framework.

In order to be able to use @PreAuthorize, the *prePostEnabled* attribute in the @EnableGlobalMethodSecurity annotation needs to be set to true.

```
@EnableGlobalMethodSecurity(prePostEnabled=true)
```

@RolesAllowed

The @RolesAllowed annotation has its origin in the JSR-250 Java security standard. This annotation is more limited than the @PreAuthorize annotation in that it only supports role-based security.

In order to use the @RolesAllowed annotation the library containing this annotation needs to be on the classpath, as it is not part of Spring Security. In addition, the *jsr250Enabled* attribute of the @EnableGlobalMethodSecurity annotation need to be set to true.

```
@EnableGlobalMethodSecurity(jsr250Enabled=true)
```

What does Spring's @Secured do?

The @Secured annotation is a legacy Spring Security 2 annotation that can be used to configured method security. It supports more than only role-based security, but does not support using Spring Expression Language (SpEL) to specify security constraints. It is recommended to use the @PreAuthorize annotation in new applications over this annotation.

Support for the @Secured annotation needs to be explicitly enabled in the @EnableGlobalMethodSecurity annotation using the *securedEnabled* attribute.

```
@EnableGlobalMethodSecurity(securedEnabled=true)
```

How are these annotations implemented?

Method-level security is accomplished using Spring AOP proxies. These have been discussed in the section on Aspect Oriented Programming.

In which security annotation are you allowed to use SpEL?

The following table shows the support for Spring Expression Language in the security annotations that can be used with Spring Security 5:

Security Annotation	Has SpEL Support?
@PreAuthorize	yes
@PostAuthorize	yes
@PreFilter	yes
@PostFilter	yes
@Secured	no
@RolesAllowed	no

References:

- Spring Security 5 API: @EnableGlobalMethodSecurity
- Spring Security 5 Reference: Method Security Expressions
- Spring Security 5 Reference: Method Security
- Spring Security 5 Reference: Method Security
- Wikipedia: JSR-250
- Java Community Process: JSR 250

Chapter 7

REST

What does REST stand for?

REST stands for REpresentational State Transfer and is an architectural style which allows clients to access and manipulate textual representations of web resources given a set of constraints.
If the application does observe the constraints, certain benefits are gained. Examples of such benefits are scalability, simplicity, portability and reliability.

References:

- Wikipedia: Representational State Transfer

What is a resource?

To quote Roy Fielding in his dissertation that, among other things, define the architectural style REST:

“Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.”

References:

- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures – REST Architectural Elements

What does CRUD mean?

The CRUD acronym stands for Create, Read, Update and Delete. These are the basic operations on REST resources, as well as persistent storage.

References:

- Wikipedia: Create, read, update and delete

Is REST secure? What can you do to secure it?

The REST architectural style in itself does not stipulate any particular security solution but suggests using a layered system style using one or more intermediaries. Security may be a responsibility of one type of intermediary.
This maps very well to what Spring has to offer when it comes to developing REST web services: A REST web service can be developed using Spring without having to consider security at all. Security can later be added to the REST web service using Spring Security, as described in the previous chapter.

While security, such as basic authentication, will make a REST service unavailable to everyone except those who know the login information, the messages passed between clients and the service will still be readable by anyone able to intercept them. To protect the messages in transit, encryption can be used such as with the HTTPS protocol.

References:

- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures – REST Architectural Elements
- Wikipedia: Representational State Transfer
- Wikipedia: HTTPS

Is REST scalable and/or interoperable?

The short answer is yes, REST web services are both scalable and interoperable.

Scalability

The statelessness, the cacheability and the layered system constraints of the REST architectural style allows for scaling a REST web service.
Statelessness ensures that requests can be processed by any node in a cluster of services without having to consider server-side state.
Cacheability allows for creating responses from cached information without the request having to proceed to the actual service, which improves network efficiency and reduces the load on the service.
A layered system allows for introducing intermediaries such as a load balancer without clients having to modify their behavior as far as sending requests to the service is concerned. The load balancer can then distribute the requests between multiple instances of the service in order to increase the request-processing capacity of the service.

Interoperability

The following elements of the REST architectural style increases interoperability:

- A REST service can support different formats for the resource representation transferred to clients and allow for clients to specify which format it wants to receive data in.
  Common formats are XML, JSON, HTML which are all formats that facilitate interoperability.
- REST resources are commonly identified using URIs, which do not depend on any particular language or implementation.
- The REST architectural style allows for a fixed set of operations on resources.
  See next section for further details.

References:

- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures – REST Architectural Elements

Which HTTP methods does REST use?

Mapping the CRUD (create, read, update and delete) operations onto HTTP verbs yields this table, which are the HTTP methods commonly used in connection to REST web services:

Operation	HTTP Verb
Create	POST
Read	GET
Update	PUT
Delete	DELETE

References:

- Wikipedia: Create, read, update and delete

What is an HttpMessageConverter?

The *HttpMessageConverter* interface specifies the properties of a converter that can perform the following conversions:

- Convert a *HttpInputMessage* to an object of specified type.
- Convert an object to a *HttpOutputMessage*.

There are many implementations of this interface that performs specific conversions. A few examples are:

- AtomFeedHttpMessageConverter
  Converts to/from Atom feeds.
- ByteArrayHttpMessageConverter
  Converts to/from byte arrays.
- FormHttpMessageConverter
  Converts to/from HTML forms.
- Jaxb2RootElementHttpMessageConverter
  Reads classes annotated with the JAXB2 annotations @XmlRootElement and @XmlType and writes classes annotated with @XmlRootElement.
- MappingJackson2HttpMessageConverter
  Converts to/from JSON using Jackson 2.x.

A *HttpMessageConverter* converts a *HttpInputMessage* created from the request to the parameter type of the controller method that is to process the request. When the controller method has finished, a *HttpMessageConverter* converts the object returned from the controller method to a *HttpOutputMessage*.

References:

- Spring 5 API Documentation: HttpMessageConverter
- Spring 5 API Documentation: HttpInputMessage
- Spring 5 API Documentation: HttpOutputMessage

Is REST normally stateless?

Statelessness of a REST service is one of the fundamental constraints of the REST architectural style. Thus REST is always stateless or else it is no longer REST.
With this said, a REST client may hold state of some kind and enclose data being part of this state in requests to a REST service. The server however is not aware of the data being part of some state and the server will not retain any such data between requests.

References:

- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures – REST Architectural Elements

What does @RequestMapping do?

The @RequestMapping annotation marks a method in a controller as a method that will be invoked to handle requests that match the configuration in the @RequestMapping annotation.
There are special cases of the @RequestMapping annotation that will match only web requests with one particular HTTP method. These annotations are:

- @GetMapping
  Matches HTTP GET requests only.
- @PostMapping
  Matches HTTP POST requests only.
- @PutMapping
  Matches HTTP PUT requests only.
- @DeleteMapping
  Matches HTTP DELETE requests only.
- @PatchMapping
  Matches HTTP PATCH requests only.

Configuration in the annotation as to decide which requests to handler are:

- consumes
  Consumable media type(s).
- headers
  Header(s).
- method
  HTTP method.
- params
  Parameter(s) and value(s) of the parameter(s).
- path
  Path mapping URI.
- produces
  Media type(s) that can be produced.

Please also refer to the section discussing how requests are mapped to controllers and controller methods earlier.

References:

- Spring 5 API Documentation: @RequestMapping
- Spring 5 API Documentation: @GetMapping

- Spring 5 API Documentation: @PostMapping
- Spring 5 API Documentation: @PutMapping
- Spring 5 API Documentation: @DeleteMapping
- Spring 5 API Documentation: @PatchMapping

Is @Controller a stereotype? Is @RestController a stereotype?

According to the first reference of this section, the @Controller is a stereotype annotation. In addition, it is located in the org.springframework.stereotype package.
The @RestController annotation is a stereotype annotation, given that its declaration is annotated with the @Controller annotation.

For further motivation on why @RestController is a stereotype annotation, see also next section.

References:

- Spring 5 Reference: @Component and further stereotype annotations
- Spring 5 Reference: Meta-annotations
- Spring 5 API Documentation: @Controller
- Spring 5 API Documentation: @RestController

What is a stereotype annotation? What does that mean?

Stereotype annotations are annotations that are applied to classes that fulfills a certain, distinct, role. The (core) Spring Framework supplies the following stereotype annotations, all found in the org.springframework.stereotype package:

- @Component
- @Controller
- @Indexed
- @Repository
- @Service

Other Spring projects does provide additional stereotype annotations.

References:

- Spring 5 Reference: @Component and further stereotype annotations
- Spring 5 API Documentation: org.springframework.stereotype package

What is the difference between @Controller and @RestController?

The @RestController annotation is an annotation that is annotated with the @Controller and the @ResponseBody annotations. That is, annotating a class with the @RestController annotation is identical to annotating the class with the @Controller and @ResponseBody annotations.

When applied at class level, the @ResponseBody annotation applies to all the methods in the controller that handles web requests.
Please refer to the next section for an explanation on the @ResponseBody annotation.

References:

- Spring 5 API Documentation: @Controller
- Spring 5 API Documentation: @RestController
- Spring 5 API Documentation: @ResponseBody

When do you need @ResponseBody?

The @ResponseBody annotation can be applied to either a controller class or a controller handler method. It causes the response to be created from the serialized result of the controller method result processed by a *HttpMessageConverter*. This is useful when you want the web response body to contain the result produced by the controller method, as is the case when implementing a backend service, for example a REST service.
The @ResponseBody annotation is not needed if the controller class is annotated with @RestController.

References:

- Spring 5 API Documentation: @ResponseBody

What are the HTTP status return codes for a successful GET, POST, PUT or DELETE operation?

HTTP response status codes are three-digit integer codes where the first digit determines the class of the response. There are five different classes of HTTP response codes:

- 1xx
  Informational. The request has been received and processing of it continues.
- 2xx
  Successful. The request has been successfully received, understood and accepted.
- 3xx
  Redirection. Further action is needed to complete the request.
- 4xx
  Client error. The request is incorrect or cannot be processed.
- 5xx
  Server error. The server failed to process what appears to be a valid request.

The following HTTP response status codes are of the successful class:

HTTP response status code 2xx	Meaning
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content

References:

- RFC 7231 Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content – Chapter 6: Response Status Codes

When do you need @ResponseStatus?

The @ResponseStatus annotation can also be used to annotate exception classes in order to specify the HTTP response status and reason that are to be returned, instead of the default Server Internal Error (500), when an exception of the type is thrown during the processing of a request in a controller handler method.

In addition, the @ResponseStatus annotation can be applied to controller handler methods in order to override the original response status information. In the annotation a HTTP response status code and a reason string can be specified. The @ResponseStatus annotation can also be applied at class level in controller classes, in which case it will apply to all the controller handler methods in the class. In both these cases, the response body will not be the response body produced by the controller handler method processing the request, instead it will be something like in this example:

```
{"timestamp":1547152877330,"status":200,"error":"OK","message":"Tea is
nice","path":"/dosomething/somethingmore"}
```

References:

- Spring 5 API Documentation: @ResponseStatus

Where do you need @ResponseBody? What about @RequestBody? Try not to get these muddled up!

@ResponseBody

As earlier, the @ResponseBody annotation is an annotation that can be applied to controller classes or controller handler methods. It is used when the web response body is to contain the result produced by the controller method(s).

@RequestBody

The @RequestBody annotation can only be applied to parameters of methods. More specific, parameters of controller handler methods. It is used when the web request body is to be bound to a parameter of the controller handler method. That is, the annotated method parameter will contain the web request body when there is a request to be handled by the controller handler method with the annotated parameter.

References:

- Spring 5 API Documentation: @ResponseBody
- Spring 5 API Documentation: @RequestBody

If you saw example Controller code, would you understand what it is doing? Could you tell if it was annotated correctly?

The listing below shows a correctly annotated REST controller that contain most of the annotations mentioned in this chapter:

```
package se.ivankrizsan.spring.mvcrestservice;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import java.util.Date;

@RestController
public class RestService {
    @RequestMapping(
        path = "dosomething/{pathVar}",
        method = RequestMethod.POST,
        consumes = "application/json",
        produces = "application/json")
    public String doSomething(
        @PathVariable(name = "pathVar", required = false) final String inPathVariable,
        @RequestBody(required = false) final String inRequestBody,
        @RequestParam(name = "param1", required = false) final String inRequestParam1,
        @RequestParam(name = "param2", required = false) final String inRequestParam2) {
        final Date theDate = new Date();
        System.out.println(
            "Received request on " + theDate + " with the request body: "
                + inRequestBody);
        System.out.println("Request param 1: " + inRequestParam1);
        System.out.println("Request param 2: " + inRequestParam2);
        return "{ doSomething: '" + theDate + "', pathVariable: '"
            + inPathVariable + "' }";
    }
}
```

Sending a POST request to the following URL, assuming the service is running on localhost, with the request body “This is the request body!” (without quotes).

```
http://localhost:8080/dosomething/somethingmore?param1=foo&param2=bar
```

The response body will contain a JSON fragment similar to this:

```
{ doSomething: 'Thu Jan 10 21:54:27 CET 2019', pathVariable: 'somethingmore' }
```

In addition, output similar to the following will appear in the console:

```
Received request on Thu Jan 10 21:54:27 CET 2019 with the request body: This is the
request body!
Request param 1: foo
Request param 2: bar
```

Note that this is a contrived example with the purpose of showing how annotations in this chapter can be used and not a good example of how this type of services should be implemented.

Do you need Spring MVC in your classpath?

The annotations discussed in this chapter, such as @RestController, @ResponseBody, @RequestBody, @PathVariable etc are present in the spring-web module that has the Maven group id org.springframework and the artifact id spring-web. In order to be able to use these annotations in a REST controller, the spring-web module must be present on the classpath.

References:

- Spring Projects @ GitHub: spring-web module annotation package

What Spring Boot starter would you use for a Spring REST application?

The Spring Boot Web Starter Maven pom.xml file contains the following in the description:

“Starter for building web, including RESTful, applications using Spring MVC.”

I would thus use the Spring Boot Web Starter if I were to implement a RESTful web service in a Spring Boot application.

References:

- Spring Boot @ GitHub: spring-boot-starter-web pom.xml

What are the advantages of the RestTemplate?

The *RestTemplate* is similar to the *JdbcTemplate* and the *JmsTemplate* in that it also implement the Template Method design pattern. The *RestTemplate* implements a synchronous HTTP client that simplifies sending requests and also enforces RESTful principles. Some advantages of the *RestTemplate* are:

- Provides a higher-level API to perform HTTP requests compared to traditional HTTP client libraries.
  Allows for sending HTTP requests with a minimum of code.
- Allows for easy selection of which underlying HTTP client library to use.
- Supports URI templates.
- Automatically encodes URI templates.
  For example, a space character in an URI will be replaced with %20 using percent-encoding.
- Supports automatic detection of content type.
- Supports automatic conversion between objects and HTTP messages.
- Supports easy customization of the HTTP message converters available to detect content type and handle conversion between objects and HTTP messages.
- Allows for easy customization of response errors.
  A custom *ResponseErrorHandler* can be registered on the *RestTemplate*.
- Allows for easy customization of URI template handling.
  This is the process of creating a URI from a URI template.
- Provides methods for conveniently sending common HTTP request types and also provides methods that allow for increased detail when sending requests.
  Examples of the former method type are: *delete*, *getForObject*, *getForEntity*, *headForHeaders*, *postForObject* and *put*.
  The latter type of methods are all called *execute* but have different parameters.

References:

- Wikipedia: Template method pattern
- Spring 5 API Documentation: RestTemplate
- Spring 5 Reference: Spring MVC – RestTemplate
- Spring 5 Reference: Remoting and web services using Spring – RestTemplate
- Wikipedia: Percent-encoding

If you saw an example using RestTemplate would you understand what it is doing?

The following are examples of performing the same GET request to the URL http://localhost:8080/persons/123/addresses/234 using a *RestTemplate* and verifying the results. The request are made against a server that always return the same response which is a response with no headers, a body containing the string “1020 Chicago Street” and the HTTP status 200.

```
    protected static String RESPONSE_BODY = "1020 Chicago Street";
    protected RestTemplate mRestTemplate;

    @Before
    public void setup() {
        mRestTemplate = new RestTemplate();
    }
    @Test
    public void getForObjectTest() {
        final String theResponseBody = mRestTemplate
            .getForObject("/persons/{personid}/addresses/{addressid}",
                String.class, "123", "234");
        Assert.assertEquals("Response body should have expected contents",
            RESPONSE_BODY, theResponseBody);
    }

    @Test
    public void getForEntityTest() {
        final ResponseEntity<String> theResponseEntity = mRestTemplate
            .getForEntity("/persons/{personid}/addresses/{addressid}",
                String.class, "123", "234");
        Assert.assertEquals("Request should be successful", HttpStatus.OK,
            theResponseEntity.getStatusCode());
        Assert.assertEquals("Response body should have expected contents",
            RESPONSE_BODY, theResponseEntity.getBody());
        Assert.assertTrue("Response should contain no headers",
            theResponseEntity.getHeaders().isEmpty());
    }

    @Test
    public void exchangeTest() {
        final URI theRequestUri = UriComponentsBuilder
            .fromUriString("/persons/{personid}/addresses/{addressid}")
            .build("123", "234");

        final RequestEntity<Void> theRequestEntity = RequestEntity
            .get(theRequestUri)
            .accept(MediaType.TEXT_PLAIN)
```

```
            .build();

        final ResponseEntity<String> theResponseEntity = mRestTemplate
            .exchange(theRequestEntity, String.class);

        Assert.assertEquals("Request should be successful", HttpStatus.OK,
            theResponseEntity.getStatusCode());
        Assert.assertEquals("Response body should have expected contents",
            RESPONSE_BODY, theResponseEntity.getBody());
        Assert.assertTrue("Response should contain no headers",
            theResponseEntity.getHeaders().isEmpty());
    }
```

The most powerful, but also least easy to understand, methods in the *RestTemplate* class are the different *exchange* methods.

References:

- Spring 5 API Documentation: RestTemplate
- Spring 5 Reference: Integration - RestTemplate

Chapter 8

Testing

Do you use Spring in a unit test?

A unit test tests one unit of functionality, a method in a class, a class or an entire module. The unit tested is tested in isolation outside of the environment for which it is intended. This means, among other things, that the Spring framework is not used to perform dependency injection in unit tests. Instead any dependencies are commonly replaced by mocks or stubs, which are created programmatically in the test-class, and set on the instance-under-test using setter-methods.

Thus Spring is not commonly used in unit tests.

References:

- Spring 5 Reference: Unit Testing
- Wikipedia: Unit testing
- Wikipedia: Mock object
- Wikipedia: Test stub
- Spring 5 Reference: Mock Objects

What type of tests typically use Spring?

Integration testing is testing of several modules of software when they are combined together and tested as a whole. In such tests the relationships between the components is significant. When using the Spring framework for dependency injection the configuration files or classes used by the dependency injection framework also need to be correct and should therefore be tested.
Thus integration tests should use Spring dependency injection.

References:

- Spring 5 Reference: Integration Testing
- Wikipedia: Integration testing

How can you create a shared application context in a JUnit integration test?

There are, in my opinion, several possible aspects of this question:

If the question is regarding how to define a Spring application context in one single place which can be used in multiple JUnit integration tests then this can be accomplished using the @ContextConfiguration annotation. This annotation is used at class level in integration test classes and allows for specifying a number of @Configuration classes, or resource locations, used to load a Spring application context. In order to define a Spring application context in one single place that is later used by multiple integration tests a class that is a common superclass to all the integration tests classes can be defined and annotated with the @ContextConfiguration annotation.
Subclasses will, as per default, inherit the @Configuration classes, resource locations and context initializers. Subclasses may append to the list of @Configuration classes or resource locations used to load a Spring application context for integration tests by specifying one or more @Configuration classes, resource locations or context initializers.

Example:

Given the following:

```
public class MyBean {
    protected String mBeanMessage;

    public MyBean(final String inBeanMessage) {
        mBeanMessage = inBeanMessage;
    }

    public String getBeanMessage() {
        return mBeanMessage;
    }
}
```

```
@Configuration
public class ContextConfigurationOne {
    @Bean
    public MyBean firstBean() {
        return new MyBean("Bean from ContextConfigurationOne");
    }
}
```

```
@Configuration
public class ContextConfigurationTwo {
```

```
    @Bean
    public MyBean secondBean() {
        return new MyBean("Bean from ContextConfigurationTwo");
    }
}
```

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ContextConfigurationOne.class)
public class TestBaseClass {
}
```

The following test, inheriting from *TestBaseClass*, will pass:

```
@ContextConfiguration(classes = ContextConfigurationTwo.class)
public class TestChildClass extends TestBaseClass {
    @Autowired
    @Qualifier("firstBean")
    protected MyBean mMyBeanOne;
    @Autowired
    @Qualifier("secondBean")
    protected MyBean mMyBeanTwo;

    @Test
    public void gotBeanFromParentTestContextConfigurationTest() {
        System.out.println("Bean instance: " + mMyBeanOne);
        assertNotNull(mMyBeanOne);
        System.out.println("Bean message: " + mMyBeanOne.getBeanMessage());
        assertTrue(mMyBeanOne.getBeanMessage().contains("ContextConfigurationOne"));
    }

    @Test
    public void gotBeanFromChildTestContextConfigurationTest() {
        System.out.println("Bean instance: " + mMyBeanTwo);
        assertNotNull(mMyBeanTwo);
        System.out.println("Bean message: " + mMyBeanTwo.getBeanMessage());
        assertTrue(mMyBeanTwo.getBeanMessage().contains("ContextConfigurationTwo"));
    }
}
```

Note that if a bean in the context configuration loaded by the child class has the same name as a bean in the context configuration loaded by the parent class, then the bean loaded by the child class will replace the bean loaded by the parent class, as is normal.

Another aspect of shared application contexts may also be caching of an application context in order to avoid recreating it in each integration test. The Spring TestContext framework will cache a *ApplicationContext* or a *WebApplicationContext* loaded in a test and it will be reused in all the subsequent tests of the same test-suite provided that the following properties of the context to be loaded matches:

- locations from the @ContextConfiguration annotation.
- classes from the @ContextConfiguration annotation.
- contextInitializerClasses from the @ContextConfiguration annotation.
- contextCustomizers from any *ContextCustomizerFactory*.
- contextLoader from the @ContextConfiguration annotation.
- parent from the @ContextHierarchy annotation.
- activeProfiles from any @ActiveProfiles annotation.
- propertySourceLocations from any @TestPropertySource annotation.
- propertySourceProperties from the @TestPropertySource annotation.
- resourceBasePath from any @WebAppConfiguration annotation.

References:

- Spring 5 Reference: Testing - Spring TestContext Framework
- Spring 5 Reference: Testing – Context management
- Spring 5 Reference: Testing – Context caching
- Spring 5 Reference: Testing – Annotations
- Spring 5 Reference: Testing – Context hierarchies
- Spring 5 API Documentation: @ContextHierarchy
- Spring 5 API Documentation: @ContextConfiguration
- Spring 5 API Documentation: ContextCustomizerFactory

When and where do you use @Transactional in testing?

The @Transactional annotation can be used in a test that alter some transactional resource, for example a database, that is to be restored to the state it had prior to the test being run.
The annotation can be applied at method level, in which case just the annotated test method(s) will run, each in its own transaction. The annotation can also be applied at class level, in which case all the test methods in the class will be executed, each in its own transaction.
A transaction in which a test-method is executed will, as default, be rolled back after the test has finished executing.

References:

- Spring 5 Reference: Testing – Transaction management

How are mock frameworks such as Mockito or EasyMock used?

Mockito and EasyMock allows for dynamic creation of mock objects that can be used to mock collaborators of class(es) under test that are external to the system or trusted.
A mock object is similar to a stub, in that it produces predetermined results when methods on the object are invoked. In addition, a mock object can also verify that it was used as expected, for example verifying method invocation sequence, parameters supplied to methods etc.
Mock objects have the advantage over stubs in that they are created dynamically and only for the specific scenario tested. Mock objects are commonly created in a test method or in a test-class before the test methods are executed.

References:

- Mockito
- EasyMock
- Martin Fowler: Mocks Aren’t Stubs

How is @ContextConfiguration used?

The purpose of the @ContextConfiguration annotation is described in its API documentation:

"@ContextConfiguration defines class-level metadata that is used to determine how to load and configure an ApplicationContext for integration tests."

The @ContextConfiguration annotates test classes and determines how the Spring application context that will be available to all tests in the class is to be loaded and configured.
The annotation has the following optional elements:

Optional Element Name	Description
classes	@Configuration classes to create application context from. Default: {}
inheritInitializers	Whether initializers from test superclasses should be inherited. Default: true
inheritLocations	Whether locations or classes (@Configuration) from test superclasses should be inherited. Default: true
initializers	Classes implementing the *ApplicationContextInitializer* interface that will be invoked to initialize the application context. Default: {}
loader	Classes implementing the *ContextLoader* interface that will be used to load the application context.
locations	Locations of XML configuration files to create application context from. Default: {}
name	Name of the context hierarchy level represented by this configuration.
value	Alias for locations.

Thus:

- The configuration from which to create the Spring application context can be either @Configuration classes (classes element) or XML configuration files (locations element).
- Initialization of the application context can be customized by specifying one or more classes implementing the *ApplicationContextInitializer* interface (initializers element).
  Initializers can also be used as an alternative to specifying @Configuration classes or XML configuration.
- Application context loading can be customized by specifying one or more classes implementing the *ContextLoader* interface.

References:

- Spring 5 API Documentation: @ContextConfiguration
- Spring 5 API Documentation: ApplicationContextInitializer
- Spring 5 API Documentation: ContextLoader
- Spring 5 Reference: Testing - @ContextConfiguration
- Spring 5 Reference: Testing – Context management

How does Spring Boot simplify writing tests?

Some of the features offered by Spring Boot that simplify writing tests are:

- Spring Boot has a starter module called spring-boot-starter-test which adds the following test-scoped dependencies that can be useful when writing tests:
  JUnit, Spring Test, Spring Boot Test, AssertJ, Hamcrest, Mockito, JSONassert and JsonPath.
- Spring Boot provides the @MockBean and @SpyBean annotations that allow for creation of Mockito mock and spy beans and adding them to the Spring application context.
- Spring Boot provide an annotation, @SpringBootTest, which allows for running Spring Boot based tests and that provides additional features compared to the Spring TestContext framework.
- Spring Boot provides the @WebMvcTest and the corresponding @WebFluxTest annotation that enables creating tests that only tests Spring MVC or WebFlux components without loading the entire application context.
- Provides a mock web environment, or an embedded server if so desired, when testing Spring Boot web applications.
- Spring Boot has a starter module named spring-boot-test-autoconfigure that includes a number of annotations that for instance enables selecting which auto-configuration classes to load and which not to load when creating the application context for a test, thus avoiding to load all auto-configuration classes for a test.
- Auto-configuration for tests related to several technologies that can be used in Spring Boot applications. Some examples are: JPA, JDBC, MongoDB, Neo4J and Redis.

References:

- Spring Boot Reference: Testing
- Spring Boot API: @SpringBootTest
- Spring Boot API: @MockBean
- Spring Boot API: @SpyBean
- Spring Boot API: @WebMvcTest
- Spring Boot API: @WebFluxTest

What does @SpringBootTest do? How does it interact with @SpringBootApplication and @SpringBootConfiguration?

When annotating a test class that run Spring Boot based tests, the @SpringBootTest annotation provide the following special features as documented in the API documentation of the annotation:

- Uses *SpringBootContextLoader* as the default *ContextLoader*.
  Provided that no other *ContextLoader* is specified using the @ContextConfiguration annotation.
- Searches for a @SpringBootConfiguration if no nested @Configuration present in the test-class, and no explicit @Configuration classes specified in the @SpringBootTest annotation.
- Allows custom *Environment* properties to be defined using the properties attribute of the @SpringBootTest annotation.
- Provides support for different web environment modes to create for the test using the webEnvironment element of the @SpringBootTest annotation.
  The following web environment modes are available: DEFINED_PORT (creates a web application context without defining a port), MOCK (creates a web application context with a mock servlet environment or a reactive web application context), NONE (creates a regular application context), RANDOM_PORT (creates a web application context and a regular server listening on a random port).
- Registers a *TestRestTemplate* and/or *WebTestClient* bean for use in web tests that are using a fully running web server.

References:

- Spring Boot Reference: Testing Spring Boot Applications
- Spring Boot API: @SpringBootTest
- Spring Boot API: SpringBootContextLoader
- Spring 5 API Documentation: ContextLoader
- Spring 5 API Documentation: @ContextConfiguration
- Spring Boot API: TestRestTemplate
- Spring 5 API Documentation: WebTestClient

Chapter 9

Spring Boot Intro

What is Spring Boot?

Spring Boot is a project gathering a number of modules under a common umbrella. Some of the more central modules are:

- spring-boot-dependencies
  Contains versions of dependencies, Maven plug-ins etc used by Spring Boot, including dependencies used by starter-modules. Contains managed dependencies for dependencies.
- spring-boot-starter-parent
  Parent pom.xml file providing dependency and plug-in management for Spring Boot applications using Maven.
- spring-boot-starters
  Parent for all the Spring Boot starter-modules.
- spring-boot-autoconfigure
  Contains the autoconfigure modules for the starters in Spring Boot.
- spring-boot-actuator
  Allows for monitoring and managing of applications created with Spring Boot.
- spring-boot-tools
  Tools used in conjunction with Spring Boot such as the Spring Boot Maven and Gradle plug-ins etc.
- spring-boot-devtools
  Developer tools that can be used when developing Spring Boot applications.

Two of the most important parts of Spring Boot are the starter and the auto-configuration modules.

Spring Boot Starters

Spring Boot starters are dependency descriptors for different technologies that can be used in Spring Boot applications. For example, if you want to use Apache Artemis for JMS messaging in your Spring Boot application, then you simply add the spring-boot-starter-artemis dependency to your application and rest assured that you have the appropriate dependencies to get you started using Artemis in your Spring Boot application.

A list of the default Spring Boot starters is available in the reference documentation as well as on GitHub.

Spring Boot Auto-Configuration

Spring Boot auto-configuration modules are modules that, like the Spring Boot starters, are concerned with a specific technology. The typical contents of an auto-configuration module is:

- One or more @Configuration class(es) that creates a set of Spring beans for the technology in question with a default configuration.
  Typically such a configuration class is conditional and require some class or interface from the technology in question to be on the classpath in order for the beans in the configuration to be created when the Spring context is created.
- A @ConfigurationProperties class.
  Allows for the use of a set of properties related to the technology in question to be used in the application's properties-file. Properties of a auto-configuration module will have a common prefix, for instance "spring.thymeleaf" for the Thymeleaf module.
  In the @ConfigurationProperties class default values for the different properties may have been assigned as to allow users of the module to get started with a minimum of effort but still be able to do some customization by only setting property values.

References:

- Spring Boot Reference: Introducing Spring Boot
- Spring Boot Reference: Creating Your Own Starter
- Spring Boot Reference: Spring Boot Actuator: Production-ready features
- Spring Boot Reference: Developer Tools
- Spring Boot Reference: Auto-configuration
- Spring Boot Reference: Starters
- Spring Boot Reference: Testing
- Spring Boot GitHub Project

What are the advantages of using Spring Boot?

Some advantages of using Spring Boot are:

- Automatic configuration of “sensible defaults” reducing boilerplate configuration.
  Configuration adapted to dependencies on the classpath so that, for example, if a HSQLDB dependency is available on the class path, then a data-source bean connecting to an in-memory HSQLDB database is created.
- Enabling getting started quickly developing an application.
  This is related to the previous step.
- Allows for easy customization when the defaults no longer suffices.
  Such customization can be accomplished by setting property values in the application’s property file and/or creating Spring beans with the same name as those created by Spring Boot in order to replace the Spring bean definitions.
- A Spring Boot project can produce an executable stand-alone JAR-file.
  Such a JAR-file can be run from the command line using the regular java -jar command and may even contain an embedded web server, for web applications, like Tomcat or Jetty.
- Provides a set of managed dependencies that have been verified to work together.
- Provides a set of managed Maven plug-ins configured to produce certain artifacts.
- Provides non-functional features commonly needed in projects.
  Some such features are security, externalized configuration, metrics and health-checks.
- Does not generate code.
- Does not require XML configuration.
- Uses well-known, standard, Spring Framework mechanisms.
  This allows developers familiar with the Spring Framework to quickly learn and use Spring Boot. This also reduce the element of surprise when adopting new technologies that are supported by Spring Boot.
- Popular in the developer community.
  There are a lot of resources on how to use develop different types of applications using Spring Boot.
- Spring Boot is a mature, well-supported and actively developed product on top of an even more mature, well-supported and actively developed framework.
- Standardize parts of application structure.
  Developers will recognize common elements when moving between different projects which use Spring Boot.

References:

- Spring Boot Reference: Introducing Spring Boot

Why is it “opinionated”?

The definition of opinionated that I have come across that I like best is:

“having strong opinions that you feel free to express”

I feel that Spring Boot “has an opinion” on how development of an application is to be done, for instance concerning the technology-related modules (starters and auto-configuration), organization of properties, configuration of modules etc.
The true accomplishment of Spring Boot is that it is opinionated but do allow for developers to customize their projects to the extent desired without becoming an obstacle.

What things affect what Spring Boot sets up?

Spring Boot detects the dependencies available on the classpath and configures Spring beans accordingly. This is accomplished using a number of annotations that allows for applying conditions to Spring configuration classes or Spring bean declaration methods in such classes.
Examples:

- A Spring bean is to be created only if a certain dependency is available on the classpath.
  Use @ConditionalOnClass and supply a class contained in the dependency in question.
- A Spring bean is to be created only if there is no bean of a certain type or with a certain name created.
  Use @ConditionalOnMissingBean and specify name or type of bean to check.

The following is a list of the condition annotations in Spring Boot and the factors that determine the result of the associated condition:

Condition Annotation	Condition Factor
@ConditionalOnClass	Presence of class on classpath.
@ConditionalOnMissingClass	Absence of class on classpath.
@ConditionalOnBean	Presence of Spring bean or bean type (class).
@ConditionalOnMissingBean	Absence of Spring bean or bean type (class).
@ConditionalOnProperty	Presence of Spring environment property.
@ConditionalOnResource	Presence of resource such as file.
@ConditionalOnWebApplication	If the application is considered to be a web application, that is uses the Spring *WebApplicationContext*, defines a session scope or has a *StandardServletEnvironment*.
@ConditionalOnNotWebApplication	If the application is not considered to be a web application.
@ConditionalOnExpression	Bean or configuration active based on the evaluation of a SpEL expression.
@ConditionalOnCloudPlatform	If specified cloud platform, Cloud Foundry, Heroku or SAP, is active.
@ConditionalOnEnabledEndpoint	Specified endpoint is enabled.
@ConditionalOnEnabledHealthIndicator	Named health indicator is enabled.
@ConditionalOnEnabledInfoContributor	Named info contributor is enabled.
@ConditionalOnEnabledResourceChain	Spring resource handling chain is enabled.
@ConditionalOnInitializedRestarter	Spring DevTools *RestartInitializer* has been applied with non-null URLs.
@ConditionalOnJava	Presence of a JVM of a certain version or within

Condition Annotation	Condition Factor
	a version range.
@ConditionalOnJndi	Availability of JNDI *InitialContext* and specified JNDI locations exist.
@ConditionalOnManagementPort	Spring Boot Actuator management port is either: Different from server port, same as server port or disabled.
@ConditionalOnRepositoryType	Specified type of Spring Data repository has been enabled.
@ConditionalOnSingleCandidate	Spring bean of specified type (class) contained in bean factory and single candidate can be determined.

References:

- Spring Boot Reference: Understanding Auto-configured Beans
- Spring Boot Reference: Condition Annotations
- Spring Boot API: @ConditionalOnClass
- Spring Boot API: @ConditionalOnBean
- Spring Boot API: @ConditionalOnMissingBean
- Spring Boot API: @ConditionalOnMissingClass
- Spring Boot Reference: Condition Annotations
- Spring Boot API

What is a Spring Boot starter POM? Why is it useful?

Please refer to the Spring Boot Starters section earlier for an explanation on what Spring Boot starter POMs are.

As earlier, the advantage of having starter POMs are that all the dependencies needed to get started with a certain technology have been gathered. A developer can rest assured that there are no dependencies missing and that all the dependencies have versions that work well together.

References:

- Spring Boot Reference: Starters

Spring Boot supports both Java properties and YML files. Would you recognize and understand them if you saw them?

A Java properties file consists of one or more rows with a key-value pair defined on each row (disregarding empty lines and comments). Example:

```
# This is a comment.
se.ivan.username=ivan
se.ivan.password=secret
city=Gothenburg
```

YML files are files containing properties in the YAML format. YAML stands for YAML Ain't Markup Language. The above properties would look like this if defined in an YML file:

```
# This is a comment.
se:
  ivan:
    username: ivan
    password: secret
city: Gothenburg
```

References:

- YAML
- Spring Boot Reference: Application Property Files

Can you control logging with Spring Boot? How?

Several aspects of logging can be controlled in different ways in Spring Boot applications.

Controlling Log Levels

As per default, messages written with the ERROR, WARN and INFO levels will be output in a Spring Boot application. To enable DEBUG or TRACE logging for the entire application, use the --debug or --trace flags or set the properties debug=true or trace=true in the application.properties file.
Log levels can be controlled at a finer granularity using the application.properties file.

```
logging.level.root=WARN
logging.level.se.ivankrizsan.spring=DEBUG
```

The above rows show how log levels can be configured in the application.properties file. The first line sets the log level of the root logger to WARN and the second line sets the log level of the se.ivankrizsan.spring logger to DEBUG.

Customizing the Log Pattern

When using the default Logback setup in a Spring Boot application, the log patterns used to write log to the console and to file can be customized using the following properties in the application.properties file:

Property Name	System Property	Use
logging.pattern.console	CONSOLE_LOG_PATTERN	Used to configure the pattern used to output log to the console.
logging.pattern.file	FILE_LOG_PATTERN	Used to configure the pattern used to output log to file.

Color-coding of Log Levels

In consoles that support ANSI, messages of different log levels can be color-coded to improve readability. This can be accomplished as shown in the following example:

```
logging.pattern.console=%clr(%d{yyyy-MM-dd HH:mm:ss}){yellow}
```

Where the color can be one of blue, cyan, faint, green, magenta, red or yellow.

Choosing a Logging System

Spring Boot uses the Commons Logging API for logging. Support for the following underlying logging frameworks is available in Spring Boot:

- Logback
  This is the default used by Spring Boot.
- Log4J2
- Java Util Logging
  The Java platform's core logging facilities.

Logging System Specific Configuration

Logging-system specific log configuration can be supplied in one of the following files depending on which logging system has been chosen:

- logback-spring.xml
  This logging configuration file allows for Logback-native configuration while also supporting the templating features of Spring Boot. Alternatively the Logback-native configuration can also be supplied in a file named logback.xml but then the Spring Boot templating features will not be available.
- log4j2.xml
  This log configuration file allows for configuring Log4J2 logging using the XML format. It is also possible to configure Log4J2 logging using YAML or JSON and then the name of the configuration files are log4j2.yaml or log4j2.yml and log4j2.json or log4j2.jsn respectively.
- logging.properties
  Configuration file used in conjunction with Java platform's core logging.

References:

- Spring Boot Reference: Log Levels
- Spring Boot Reference: Custom Log Configuration
- Spring Boot Reference: Console output
- Spring Boot Reference: Logging
- JavaSE 10 API: Package java.util.logging

Where does Spring Boot look for property file by default?

The default properties of a Spring Boot application are stores in the application's JAR in a file named "application.properties". When developing, this file is found in the src/main/resources directory.
Individual property values defined in this application.properties file can be customized using for example command-line arguments when starting the application. The default properties can be overridden in their entirety using an external application.properties file or a YAML equivalent.

Another example is with a profile-specific properties file that contain a subset of the application properties. When the profile is activated, the properties in the profile-specific property file will override the ones in the default properties file (application.properties).

Spring Boot looks for property files in the following locations in the order listed:

- When Spring Boot Devtools is active, the Spring Boot Devtools properties file in the home directory of the current user (~/.spring-boot-devtools.properties).
- In a test, at the location specified using the @TestPropertySource annotation.
- In a profile-specific application properties file outside of the application JAR with the name application-{profile name}.properties and the YAML counterparts.
- In a profile-specific application properties file inside of the application JAR with the name application-{profile name}.properties and the YAML counterparts.
- Application properties outside of the application JAR with the name application.properties and the YAML counterparts.
- Application properties inside of the application JAR with the name application.properties and the YAML counterparts.
- One or more locations as specified in the @PropertySource annotation annotating a @Configuration class.

In addition to the above, there are several ways to specify individual property values and a certain order in which these are evaluated. Please consult the section on Externalized Configuration in the Spring Boot Reference for further information.

References:

- Spring Boot Reference: Externalized Configuration
- Spring Boot Reference: Developer Tools, Global Settings
- Spring 5 API Documentation: @TestPropertySource
- Spring 5 API Documentation: @PropertySource
- YAML

How do you define profile specific property files?

Properties specific to a profile can, as before, be a subset of the properties of the application properties and, for instance, configure the application to run in a certain environment. Such profile specific properties can be defined in properties-files with the following name pattern:

application-{profile}.properties

References:

- Spring Boot Reference: Profile-specific Properties

How do you access the properties defined in the property files?

As described earlier, the @Value annotation can be used to inject property values into Spring beans and configuration classes.

In addition, properties can be bound to Java bean class(es) using the @ConfigurationProperties annotation on the Java bean class(es) and the @EnableConfigurationProperties annotation on one @Configuration class. This allows for automatic validation of property values, since the setter methods of such bean classes only accept a certain type. A drawback of this approach is that SpEL expressions in property values are not evaluated.

References:

- Spring 5 API Documentation: @Value
- Spring Boot Reference: Type-safe Configuration Properties
- Spring Boot API: @ConfigurationProperties
- Spring Boot API: @EnableConfigurationProperties

What properties do you have to define in order to configure external MySQL?

To connect to an external MySQL database, the following properties need to be set in the application.properties file:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=username
spring.datasource.password=secret
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

References:

- Spring Boot Reference: Connection to a Production Database

How do you configure default schema and initial data?

Configure Default Schema

The name of a default schema can be supplied in the datasource URL, as shown in the following example:

```
spring.datasource.url=jdbc:postgresql://localhost/databasename?
currentSchema=defaultschemaname
```

Note that in some databases, MySQL for example, database is equivalent to schema and the default schema is specified by the “databasename” part of the datasource URL in the example above. A datasource URL for MySQL would thus look like this:

```
spring.datasource.url=jdbc:mysql://localhost/defaultschemaname
```

There are other ways of accomplishing the same for specific technologies, such as Hibernate in this example:

```
spring.jpa.properties.hibernate.default_schema=defaultschemaname
```

However, including the name of the default schema in the datasource URL is to be preferred since it does not create any coupling to the underlying JPA library – Hibernate in this example.

Database Initialization

If using JPA, then setting the vendor-independent property spring.jpa.generate-ddl to true will cause JPA to initialize the application’s database, creating tables etc.
Using vendor-specific properties, like Hibernate’s spring.jpa.hibernate.ddl-auto, may allow for more control over database initialization, depending on the vendor.

If plain JDBC is used, Spring Boot will execute SQL files named according to the following naming patterns at application startup to initialize the database:

```
schema.sql
schema-${platform}.sql
```

Where *platform* is the value of the property spring.datasource.platform.

Supplying Initial Data

When a Spring Boot application is started, Spring Boot executes SQL files named according to the following naming conventions to allow for insertion of initial data into the database:

```
data.sql
data-${platform}.sql
```

As before, *platform* is the value of the property spring.datasource.platform.
This method of supplying initial data can be used in projects that uses JPA as well as those who do not.

References:

- Spring Boot Reference: Database Initialization
- Spring Boot Reference: Connection to a Production Database

What is a fat jar? How is it different from the original jar?

A “fat JAR” is a self-contained JAR file which contains all dependencies needed for the application to be run. The JAR files of the dependencies are contained in the application’s JAR file. With the current version of Spring Boot, the nested JAR files of the application’s dependencies are contained in the BOOT-INF/lib directory.

The Java Tutorial states that JAR files typically contain the class files of and auxiliary resources of an application. The difference of such a JAR file compared to a fat JAR is the dependencies, which with the original JAR had to be located outside of the JAR file.

References:

- Spring Boot Reference: Packaging Your Application for Production
- Spring Boot Reference: Creating an Executable Jar
- Spring Boot Reference: Appendix E - The Executable Jar Format
- The Java Tutorials: Packaging Programs in JAR Files

What is the difference between an embedded container and a WAR?

An embedded container is packaged in the application JAR-file and will contain only one single application. A WAR-file will need to be deployed to a web container, such as Tomcat, before it can be used. The web container to which the WAR-file is deployed may contain other applications.

What embedded containers does Spring Boot support?

Spring Boot supports the following embedded containers:

- Tomcat
- Jetty
- Undertow

References:

- Spring Boot Reference: Embedded Servlet Container Support

Chapter 10

Spring Boot Auto-Configuration

How does Spring Boot know what to configure?

Please refer to the section What things affect what Spring Boot sets up? earlier.

What does @EnableAutoConfiguration do?

The @EnableAutoConfiguration annotation enables Spring Boot auto-configuration. As earlier, Spring Boot auto-configuration attempts to create and configure Spring beans based on the dependencies available on the class-path to allow developers to quickly get started with different technologies in a Spring Boot application and reducing boilerplate code and configuration.

References:

- Spring Boot API: @EnableAutoConfiguration
- Spring Boot Reference: The @EnableAutoConfiguration Annotation
- Spring Boot Reference: Auto-configuration

What does @SpringBootApplication do?

The @SpringBootApplication is a convenience-annotation that can be applied to Spring Java configuration classes. The @SpringBootApplication is equivalent to the three annotations @Configuration, @EnableAutoConfiguration and @ComponentScan.

References:

- Spring Boot API: @SpringBootApplication
- Spring 5 API Documentation: @Configuration
- Spring Boot API: @EnableAutoConfiguration
- Spring 5 API Documentation: @ComponentScan

Does Spring Boot do component scanning? Where does it look by default?

Spring Boot does not do component scanning unless a configuration class, annotated with @Configuration, that is also annotated with the @ComponentScan annotation or an annotation, for instance @SpringBootApplication, that is annotated with the @ComponentScan annotation.

The base package(s) which to scan for components can be specified using the *basePackages* element in the @ComponentScan annotation or by specifying one or more classes that are located in the base package(s) to scan for components by using the *basePackageClasses* element.
If none of the above elements are used, component scanning will take place using the package in which the configuration class annotated with @ComponentScan as the base package.

References:

- Spring Boot API: @SpringBootApplication
- Spring 5 API Documentation: @ComponentScan

What is spring.factories file for?

The spring.factories file can be used to:

- Register application event listeners regardless of how the Spring Boot application is created (configured).
  Implement a class that inherits from *SpringApplicationEvent* and register it in the spring.factories file.
- Locate auto-configuration candidates in, for instance, your own starter module.
- Register a filter to limit the auto-configuration classes considered.
  See *AutoConfigurationImportFilter*.
- Activate application listeners that creates a file containing the application process id and/or creates file(s) containing the port number(s) used by the running web server (if any).
  These listeners, *ApplicationPidFileWriter* and *WebServerPortFileWriter*, both implement the *ApplicationListener* interface.
- Register failure analyzers.
  Failure analyzers implement the *FailureAnalyzer* interface and can be registered in the spring.factories file.
- Customize the environment or application context prior to the Spring Boot application has started up.
  Classes that implementing the *ApplicationListener*, *ApplicationContextListener* or the *EnvironmentPostProcessor* interfaces may be registered in the spring.factories file.
- Register the availability of view template providers.
  See the *TemplateAvailabilityProvider* interface.

References:

- Spring Boot Reference: Application and Event Listeners
- Spring Boot API: SpringApplicationEvent
- Spring Boot Reference: Understanding Auto-configured Beans
- Spring Boot API: AutoConfigurationImportFilter
- Spring Boot Reference: Locating Auto-configuration Candidates
- Spring Boot Reference: Process Monitoring
- Spring Boot API: ApplicationPidFileWriter
- Spring Boot API: WebServerPortFileWriter
- Spring Boot Reference: Create Your Own Failure Analyzer

- Spring Boot API: FailureAnalyzer
- Spring Boot Reference: Customize the Environment or ApplicationContext Before It Starts
- Spring 5 API Documentation: ApplicationListener
- Spring 5 API Documentation: ApplicationContextInitializer
- Spring Boot API: EnvironmentPostProcessor
- Spring Boot Repository – spring.factories file
- Spring Boot API: TemplateAvailabilityProvider

How do you customize Spring auto-configuration?

The simplest way to customize Spring auto-configuration is by changing any property values used by the related beans.

The next option is to create a Spring bean to replace a bean created by the auto-configuration. To replace the auto-configuration bean, the new bean needs to have a matching type and/or name (depending on the @Conditional annotation(s) annotating the bean method in the auto-configuration). Note that in order to be able to override bean definitions using beans with the same name as the original bean, the following application property needs to be set to true in the Spring Boot application:

```
spring.main.allow-bean-definition-overriding=true
```

Another option is to disable one or more auto-configuration classes altogether. This can be accomplished either in the @EnableAutoConfiguration annotation, by specifying the class(es) or fully qualified class name(s), or by using the *spring.autoconfigure.exclude* property with one or more fully qualified class name(s).

References:

- Spring Boot Reference: Auto-configuration
- Spring Boot API: @EnableAutoConfiguration

What are examples of @Conditional annotations? How are they used?

Please refer to the section What things affect what Spring Boot sets up? earlier for a list of the available @Conditional annotations.

@Conditional annotations are typically used in auto-configuration classes such as those of Spring Boot starter modules.
@Conditional annotations are applied either at class level in @Configuration classes or at method level, also in @Configuration classes, annotating @Bean methods. They allow for the creation of Spring beans to be conditional depending on, for example, the presence of a certain class on the classpath.
@Conditional annotations also allow for conditional creation of Spring beans if a bean of a certain type is not already present in the application context. This allows for, for example, the creation of Spring Boot starter modules that contain a set of default beans that can be replaced.

References:

- Spring Boot Reference: Condition Annotations

Chapter 11

Spring Boot Actuator

What values does Spring Boot Actuator provide?

I interpret the question as what valuable/useful features does Spring Boot Actuator provide when being added to a Spring Boot application. In addition, the information in this section also provide an overview of the different categories of information that Spring Boot Actuator, in its basic version without extensions, can provide about an application.

Spring Boot Actuator provides the following features which can be automatically applied to Spring Boot applications:

- Application health monitoring
- Application metrics
- Management
- Auditing

The following built-in Spring Boot Actuator endpoints are available:

Endpoint id	Description
auditevents	Application auditing events.
beans	Lists the Spring beans in the application.
conditions	Conditions evaluated and the results in connection to configuration and auto-configuration.
configprops	Lists all @ConfigurationProperties.
env	Lists properties from the Spring ConfigurableEnvironment.
flyway	Lists applied Flyway database migrations.
health	Application health information.
httptrace	Lists HTTP trace information about the latest HTTP request-response exchanges.
info	Arbitrary application information.
loggers	Allows for viewing and modification of the application’s log configuration.
liquibase	Lists applied Liquibase database migrations.
metrics	Application metrics information.
mappings	Lists the @RequestMapping paths of the application.
scheduledtasks	Lists the scheduled tasks of the application.
sessions	Retrieval and deletion of user sessions from a Spring Session backend store. Not available

	with reactive web applications.
shutdown	Shutdown of the application.
threaddump	Exposes the result of a thread-dump.

In web applications, that is applications using Spring MVC, Spring WebFlux or Jersey, the following additional endpoints are also available:

Endpoint id	Description
heapdump	Performs a heap dump and returns the heap dump file.
jolokia	Exposes JMX beans over HTTP. Not available with Spring WebFlux.
logfile	Allows retrieval of whole or part of the log file as specified by logging.file or logging.path properties.
prometheus	Exposes metrics in a format that can be consumed by the Prometheus server.

With a Spring Boot application in which Spring Boot Actuator is enabled and with the following two properties set to the values shown, the available Actuator endpoints can be listed by sending a HTTP GET request, for example in a browser, to the http://localhost:8080/actuator/ URL.

```
management.endpoints.enabled-by-default=true
management.endpoints.web.exposure.include=*
```

References:

- Spring Boot Reference: Spring Boot Actuator
- Spring Boot Reference: Spring Boot Actuator – Endpoints

What are the two protocols you can use to access actuator endpoints?

Spring Boot Actuator endpoints can be accessed over HTTP and JMX, provided that an endpoint is enabled. An endpoint also need to be accessible over the protocol in question - some endpoints, like the jolokia endpoint, are not accessible over a certain protocol (JMX in the case of the jolokia endpoint).

References:

- Spring Boot Reference: Spring Boot Actuator – Endpoints
- Spring Boot Reference: Spring Boot Actuator – Monitoring and Management Over HTTP
- Spring Boot Reference: Spring Boot Actuator – Monitoring and Management Over JMX

How do you access an endpoint using a tag?

Tags can be added as query parameters to the end of an URL accessing a Spring Boot Actuator endpoint in order to further refine the request.

The following example requests the amount of used memory in the G1 Old Gen part of the heap (in certain cases the G1 Old Gen will not be present but there will be a PS Old Gen instead):

```
http://localhost:8080/actuator/metrics/jvm.memory.used?tag=area:heap&tag=id:G1%20Old
%20Gen
```

When viewed in a browser, the raw JSON response to the above request looks like this:

```
{
  "name": "jvm.memory.used",
  "description": "The amount of used memory",
  "baseUnit": "bytes",
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 24403456
    }
  ],
  "availableTags": []
}
```

The next example will give the sum of the used heap memory in the G1 Eden Space, G1 Survivor Space and G1 Old Gen. In certain cases the “G1” in the names of above ids will be replaced by “PS”.

Example:

```
http://localhost:8080/actuator/metrics/jvm.memory.used?tag=area:heap
```

When issued in a browser, the raw JSON response to the above request looks like this:

```
{
  "name": "jvm.memory.used",
  "description": "The amount of used memory",
  "baseUnit": "bytes",
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 31743488
    }
  ],
  "availableTags": [
    {
      "tag": "id",
```

```
      "values": [
        "G1 Old Gen",
        "G1 Survivor Space",
        "G1 Eden Space"
      ]
    }
  ]
}
```

References:

- Micrometer Concepts: Naming meters
- Spring Boot Reference: Spring Boot Actuator – Metrics Endpoint

What is metrics for?

Metrics are measurements of, in the case of a Spring Boot application, different aspects of the application with the purpose of determining for instance the performance of the application at different points in time under different conditions.

Some examples of types of metrics are:

- Timers
- Gauges
- Counters
- Distribution summaries
- Long task timers

Some examples of metrics that can be found in a Spring Boot application are:

- Response time of HTTP requests.
- Number of active connections in a database connection pool.
- Memory usage.
  This can be for instance heap memory usage.
- Garbage collection statistics.

References:

- Micrometer Concepts: Meters
- Micrometer Concepts: Counters
- Micrometer Concepts: Gauges
- Micrometer Concepts: Timers
- Micrometer Concepts: Distribution Summaries
- Micrometer Concepts: Long Task Timers

How do you create a custom metric with or without tags?

The following example shows a component that registers two metrics; the first metric is without tags and the second metric has one single tag.

```
package se.ivankrizsan.spring;

import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.Tags;
import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.List;

@Component
public class MyComponentWithMetrics {
    protected Long mMetricsNumber = 12345L;
    protected List<String> mFruitList = new ArrayList<>();

    public MyComponentWithMetrics(final MeterRegistry inMeterRegistry) {
        mFruitList.add("banana");
        mFruitList.add("guava");
        mFruitList.add("lemon");
        mFruitList.add("orange");

        /* Register a custom metrics without tags that contains a long number. */
        inMeterRegistry.gauge(
            "mycomponent.longnumber",
            Tags.empty(),
            mMetricsNumber);

        /* Register a custom metrics with one tag that represents the size of the fruit-
list. */
        inMeterRegistry.gaugeCollectionSize(
            "mycomponent.fruitlist.size",
            Tags.of("id", "medium"),
            mFruitList);
    }
}
```

When viewed in a browser, the first metric, present at the URL localhost:8080/actuator/metrics/mycomponent.longnumber, yields the following JSON representation:

```
{
  "name": "mycomponent.longnumber",
  "description": null,
  "baseUnit": null,
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 12345
    }
  ],
  "availableTags": []
}
```

The second metric, present at the URL http://localhost:8080/actuator/metrics/mycomponent.fruitlist.size, has the following JSON representation:

```
{
  "name": "mycomponent.fruitlist.size",
  "description": null,
  "baseUnit": null,
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 4
    }
  ],
  "availableTags": [
    {
      "tag": "id",
      "values": [
        "medium"
      ]
    }
  ]
}
```

Rererences:

- Spring Boot Reference: Spring Boot Actuator – Registering custom metrics

What is Health Indicator?

A Spring Boot Actuator health indicator provides an indication of health of a certain component or subsystem, such as a database, a message broker etc. A health indicator contains the logic to perform health checks and store the result of the health check in the health indicator, from which it can be retrieved.

References:

- Spring Boot Reference: Spring Boot Actuator – Health Information
- Spring Boot API: HealthIndicator
- Spring Boot API: Health

What are the Health Indicators that are provided out of the box?

The following health indicators are available and are auto-configured by Spring Boot when appropriate (for example when the relevant library is on the classpath and the necessary configuration is provided).

Health Indicator	Description
CassandraHealthIndicator	Verifies the availability of a Cassandra database server.
DiskSpaceHealthIndicator	Verifies that available disk space is above a certain threshold.
DataSourceHealthIndicator	Verifies the status of a *DataSource* and optionally executes a test query.
ElasticsearchHealthIndicator	Verifies the availability of an Elasticsearch cluster.
ElasticsearchJestHealthIndicator	Verifies the availability of an Elasticsearch cluster using a JestClient.
InfluxDbHealthIndicator	Verifies the availability of an Influx database server.
JmsHealthIndicator	Verifies the availability of a JMS message broker.
LdapHealthIndicator	Verifies the availability of LDAP server(s).
MailHealthIndicator	Verifies the availability of a mail server.
MongoHealthIndicator	Verifies the availability of a Mongo database server.
Neo4jHealthIndicator	Verifies the availability of a Neo4J database server.
RabbitHealthIndicator	Verifies the availability of a RabbitMQ message broker.
RedisHealthIndicator	Verifies the availability of a Redis data store.
SolrHealthIndicator	Verifies the availability of a Apache Solr server.

References:

- Spring Boot Reference: Spring Boot Actuator – Health Information
- Spring Boot API: Health
- Spring Boot API: CassandraHealthIndicator
- Spring Boot API: DiskSpaceHealthIndicator

- Spring Boot API: DataSourceHealthIndicator
- Spring Boot API: ElasticsearchHealthIndicator
- Spring Boot API: ElasticsearchJestHealthIndicator
- Spring Boot API: InfluxDbHealthIndicator
- Spring Boot API: JmsHealthIndicator
- Spring Boot API: LdapHealthIndicator
- Spring Boot API: MailHealthIndicator
- Spring Boot API: MongoHealthIndicator
- Spring Boot API: Neo4jHealthIndicator
- Spring Boot API: RabbitHealthIndicator
- Spring Boot API: RedisHealthIndicator
- Spring Boot API: SolrHealthIndicator

What is the Health Indicator status?

Spring Boot Actuator health indicator status expresses the state of a component or subsystem. Health has a status which consists of a code and a description, both stored as strings.

References:

- Spring Boot API: Health
- Spring Boot API: Status

What are the Health Indicator statuses that are provided out of the box?

There are four predefined health indicator statuses in Spring Boot:

Status	Description
Status.DOWN	Component or subsystem is malfunctioning.
Status.OUT_OF_SERVICE	Component or subsystem has been taken out of service and should not be used.
Status.UNKNOWN	Status of component or subsystem is not known.
Status.UP	Component or subsystem is functioning as expected.

References:

- Spring Boot API: Status

How do you change the Health Indicator status severity order?

The severity order of the status codes can be changed and new health indicator status codes can be added using the management.health.status.order configuration property.
The following example shows a new status with code FATAL added to the existing status codes:

```
management.health.status.order=FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

If the health endpoint is accessed over HTTP, a mapping from HTTP status code to health status can be created using another configuration property. To map the status code 418, “I’m a teapot”, to the above FATAL status code, the following property would be added to the application’s properties:

```
management.health.status.http-mapping.FATAL=418
```

References:

- Spring Boot Reference: Spring Boot Actuator – Writing Custom HealthIndicators

Why do you want to leverage 3rd-party external monitoring system?

There are several reasons for using a third-party external monitoring system in combination with Spring Boot Actuator. Some of these reasons are:

- Gather data from multiple applications in one place.
- Retain monitoring data over time.
  This gives several subsequent opportunities, some of which are listed below.
- Allow for querying monitoring data.
- Allow for visualization of monitoring data.
- Enable alerting based on monitoring data.
- Allow for analysis of monitoring data to find trends and to discover anomalies.

Chapter 12

Spring Boot Testing

When do you want to use @SpringBootTest annotation?

When annotating a test class that run Spring Boot based tests, the @SpringBootTest annotation provide the following special features as documented in the API documentation of the annotation:

- Uses *SpringBootContextLoader* as the default *ContextLoader*.
  Provided that no other *ContextLoader* is specified using the @ContextConfiguration annotation.
- Searches for a @SpringBootConfiguration if no nested @Configuration present in the test-class, and no explicit @Configuration classes specified in the @SpringBootTest annotation.
- Allows custom *Environment* properties to be defined using the properties attribute of the @SpringBootTest annotation.
- Provides support for different web environment modes to create for the test using the webEnvironment element of the @SpringBootTest annotation.
  The following web environment modes are available: DEFINED_PORT (creates a web application context without defining a port), MOCK (creates a web application context with a mock servlet environment or a reactive web application context), NONE (creates a regular application context), RANDOM_PORT (creates a web application context and a regular server listening on a random port).
- Registers a *TestRestTemplate* and/or *WebTestClient* bean for use in web tests that are using a fully running web server.

Thus you use the @SpringBootTest annotation when you want some or all of the above.

References:

- Spring Boot Reference: Testing Spring Boot Applications
- Spring Boot API: @SpringBootTest
- Spring Boot API: SpringBootContextLoader
- Spring 5 API Documentation: ContextLoader
- Spring 5 API Documentation: @ContextConfiguration
- Spring Boot API: TestRestTemplate
- Spring 5 API Documentation: WebTestClient

What does @SpringBootTest auto-configure?

@SpringBootTest auto-configures the following:

- The default context loader.
  Will be set to *SpringBootContextLoader* if a context loader is not configured in the application.
- A *TestRestTemplate* and/or *WebTestClient*.
  For use in web tests that are using a fully running web server.
  A *TestRestTemplate* is only created and configured when WebEnvironment.RANDOM_PORT or WebEnvironment.DEFINED_PORT is specified in the @SpringBootTest annotation.
- Configuration as specified by class annotated with @SpringBootConfiguration or nested @Configuration present in the test-class.
  Only if no explicit @Configuration classes specified in the @SpringBootTest annotation.
- A PropertySourcesPlaceholderConfigurer.

References:

- Spring Boot Reference: ConfigFileApplicationContextInitializer
- Spring Boot Reference: Testing Spring Boot Applications

What dependencies does spring-boot-starter-test brings to the classpath?

The spring-boot-starter-test starter adds the following test-scoped dependencies to the classpath:

- JUnit
  Unit-testing framework.
- Spring Test and Spring Boot Test
- AssertJ
  Fluent assertions for Java.
- Hamcrest
  Framework for writing matchers that are both powerful and easy to read.
- Mockito
  Mocking framework for Java.
- JSONassert
  Tools for verifying JSON representation of data.
- JsonPath
  A Java DSL for reading JSON documents.

References:

- Spring Boot Reference: Testing – Test Scope Dependencies

How do you perform integration testing with @SpringBootTest for a web application?

Four different types of web environments can be specified using the webEnvironment attribute of the @SpringBootTest annotation:

- MOCK
  Loads a web *ApplicationContext* and provides a mock web environment. Does not start a web server.
- RANDOM_PORT
  Loads a *WebServerApplicationContext*, provides a real web environment and starts an embedded web server listening on a random port. The port allocated can be obtained using the @LocalServerPort annotation or @Value("${local.server.port}").
  Web server runs in a separate thread and server-side transactions will not be rolled back in transactional tests.
- DEFINED_PORT
  Loads a *WebServerApplicationContext*, provides a real web environment and starts an embedded web server listening on the port configured in the application properties, or port

8080 if no such configuration exists.
Web server runs in a separate thread and server-side transactions will not be rolled back in transactional tests.

- NONE
  Loads an *ApplicationContext* without providing any web environment.

In the test class, annotated with @SpringBootTest, a *TestRestTemplate* and/or *WebTestClient* can be injected and used to send requests either to the mock web environment or to the embedded web server.

References:

- Spring Boot Reference: Testing Spring Boot Applications
- Spring Boot API: WebServerApplicationContext
- Spring Boot API: @LocalServerPort
- What does @SpringBootTest auto-configure?

When do you want to use @WebMvcTest? What does it auto-configure?

The @WebMvcTest annotation is intended to be used in tests which aim are to test only Spring MVC components, disabling full auto-configuration and only applying configuration relevant to testing of MVC components. Thus this annotation is not suitable for integration tests.

The @WebMvcTest annotation auto-configures the following:

- Spring Security
- MockMvc
- Caching
- Message source
  Support for resolving messages typically found in resource bundles, including parameterization and internationalization.
- FreeMarker
  A templating engine.
- Groovy templates
- Gson
  A library to create JSON representation from Java objects and vice versa.
- Hypermedia
  Spring HATEOAS.
- HTTP message converters
- Jackson
  Another library to create JSON representation from Java objects and vice versa.
- JSON-B
  Another library to create JSON representation from Java objects and vice versa.
- Mustache
  Another templating engine.
- Thymeleaf
  Another templating engine.
- JSR-303 bean validation
- Web MVC
- Web MVC error rendering

- WebClient mock MVC integration
- Selenium WebDriver mock MVC integration

References:

- Spring Boot Reference: Appendix D – Test auto-configuration annotations
- Spring Boot API: @WebMvcTest
- Spring 5 API Documentation: MockMvc

What are the differences between @MockBean and @Mock?

Both the @MockBean and @Mock annotation can be used to create Mockito mocks but there are some differences between the two annotations:

- @Mock can only be applied to fields and parameters while @MockBean can only be applied to classes and fields.
- @Mock can be used to mock any Java class or interface while @MockBean only allows for mocking of Spring beans or creation of mock Spring beans.
  @MockBean can be used to mock existing beans but also to create new beans that will belong to the Spring application context.
- To be able to use the @MockBean annotation, the Spring runner (@RunWith(SpringRunner.class)) has to be used to run the associated test.
- @MockBean can be used to create custom annotations for specific, reoccurring, needs.

References:

- Spring Boot API: @MockBean
- Spring Boot Reference: Mocking and Spying Beans
- Mockito JavaDoc: @Mock

When do you want @DataJpaTest for? What does it auto-configure?

The @DataJpaTest annotation is used to annotate test-classes that contain tests of only JPA components.

The @DataJpaTest annotation auto-configures the following:

- Caching
- Spring Data JPA repositories
- Flyway database migration tool
- A *DataSource*
  The data-source will, as default, use an embedded in-memory database (test database).
- Data source transaction manager
  A transaction manager for a single *DataSource*.
- A *JdbcTemplate*
- Liquibase database migration tool
- JPA base configuration for Hibernate
- Spring transaction
- A test database
- A JPA entity manager for tests

References:

- Spring Boot Reference: Appendix D – Test auto-configuration annotations
- Spring Boot Reference: Auto-configured Data JPA Tests
- Spring Boot API: @DataJpaTest