

What is Spring Framework?

Spring is one of the most widely used Java EE framework. Spring framework core concepts are “Dependency Injection” and “Aspect Oriented Programming”.

Spring framework can be used in normal java applications also to achieve loose coupling between different components by implementing dependency injection and we can perform cross cutting tasks such as logging and authentication using spring support for aspect oriented programming.

I like spring because it provides a lot of features and different modules for specific tasks such as Spring MVC and Spring JDBC. Since it's an open source framework with a lot of online resources and active community members, working with Spring framework is easy and fun at same time.

What is loose coupling and tight coupling?

How does Spring framework works?

So how does spring work internally?

In spring everything is a single [POJO](#) and it works almost like a glorified [HashMap](#).

You can also imagine that as a **Registry**. Surely it doing much more than a **HashMap** and shoving object in there, but there aren't a lot of magic happening.

The spring framework itself is built on a really simple architecture and aren't going on too much behind the scenes.

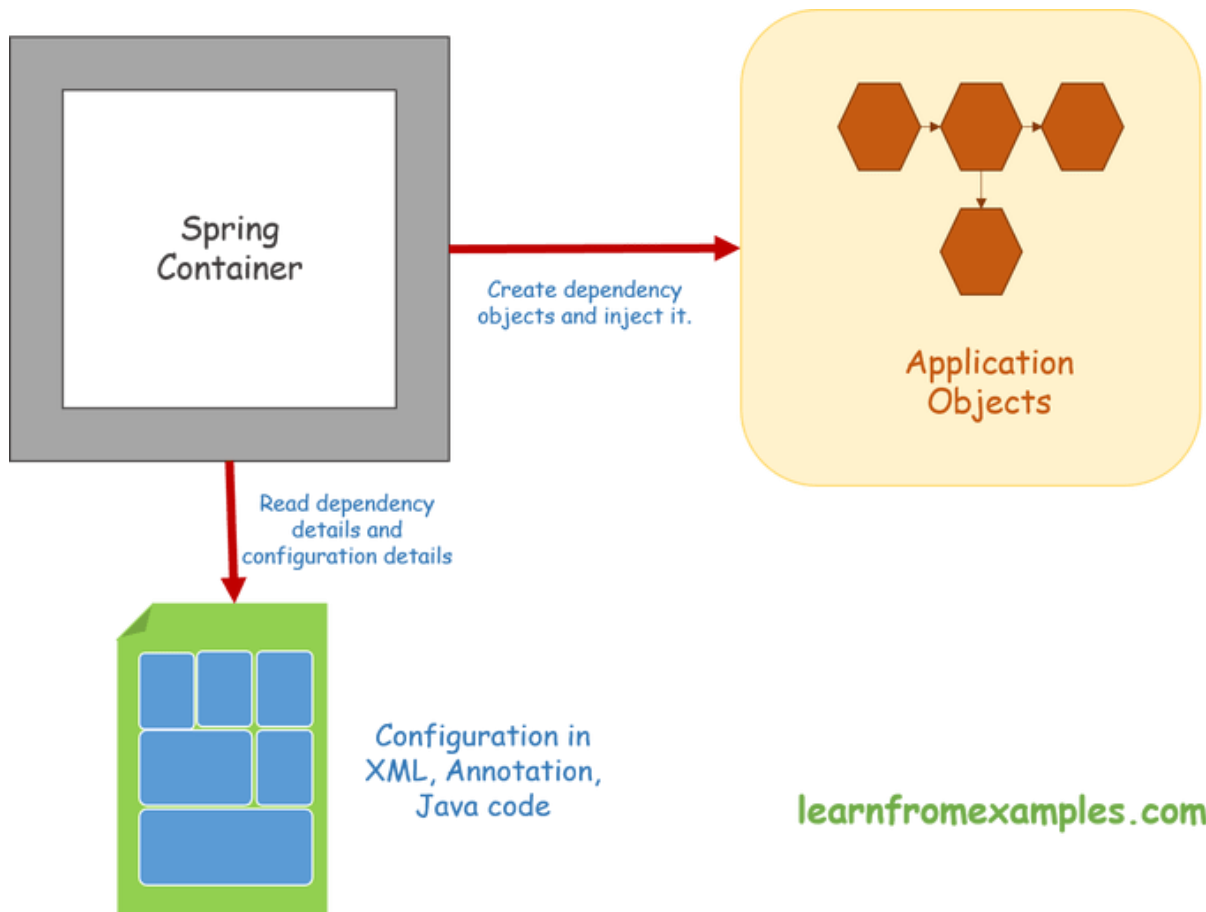
It's really simple to try it, you can write a simple main method and just run it.

The Spring IoC container

It is the heart of the Spring Framework. The IoC container receives metadata from either an **XML file, Java annotations, or Java code**. The container gets its instructions on what objects to instantiate, configure, and assemble from simple Plain Old Java Objects (POJO) by reading the configuration metadata provided. These created objects through this process called **Spring Beans**.

The responsibilities of IoC container are:

1. Instantiating the bean
2. Wiring the beans together
3. Configuring the beans
4. Managing the bean's entire life-cycle



This diagram represents an abstract view of the working of Spring Framework. It shows how Spring makes use of Java POJO classes and configuration metadata.

The real benefits of Spring are showing up through its wiring constructs and using auto-wiring.

If you look at the picture above, you can see that there is a lot of squares in there.

That's how the spring container would look like if you graphed it all out.

The little squares are the **Spring beans** and you can see **their references**. Some of them are standalone, some of them are referencing other beans.

Just how they wired up and how it makes all these objects that we're using altogether.

The JDBC Example

According to our example of a JDBC connection that we [were looking at](#), one of these might be the entity manager, one of them might be a statement or a prepared statement. The other one can be a connection behind the scenes.

All of our beans are get stored in this container, and then we access them out of that. This is where the HashMap or Registry metaphor comes into the play. We can simply access them in appropriate ways.

What are principals of Spring Framework?

What is life cycle of Spring Framework?

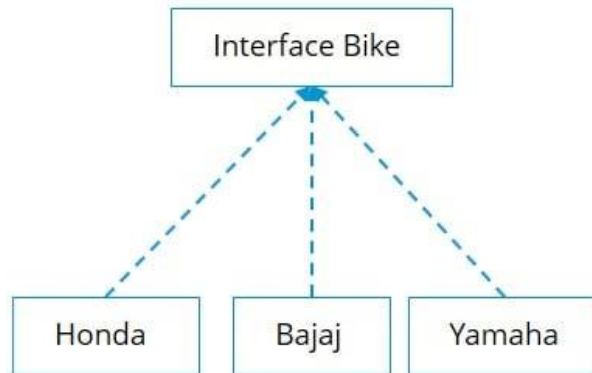
Why Spring is Simplicity?

Spring Framework is simple because its non-invasive as it uses POJO and POJI models. <>

- POJO (Plain Old Java Objects): A Java class not coupled with any technology or any framework is called "POJO".
- POJI (Plain Old Java Interfaces): A Java interface not coupled with any technology or any framework is called "POJI".

Loose Coupling : Spring Framework is loosely coupled because it has concepts like Dependency Injection, AOP etc. These features help in reducing dependency and increasing the modularity within the code. Lets understand this with an example.

Here I have a Bike interface which has a start() method. It is further implemented by three classes,



namely : Yamaha, Honda and Bajaj.

```
1 public interface Bike
2 {
3     public void start();
4 }
```

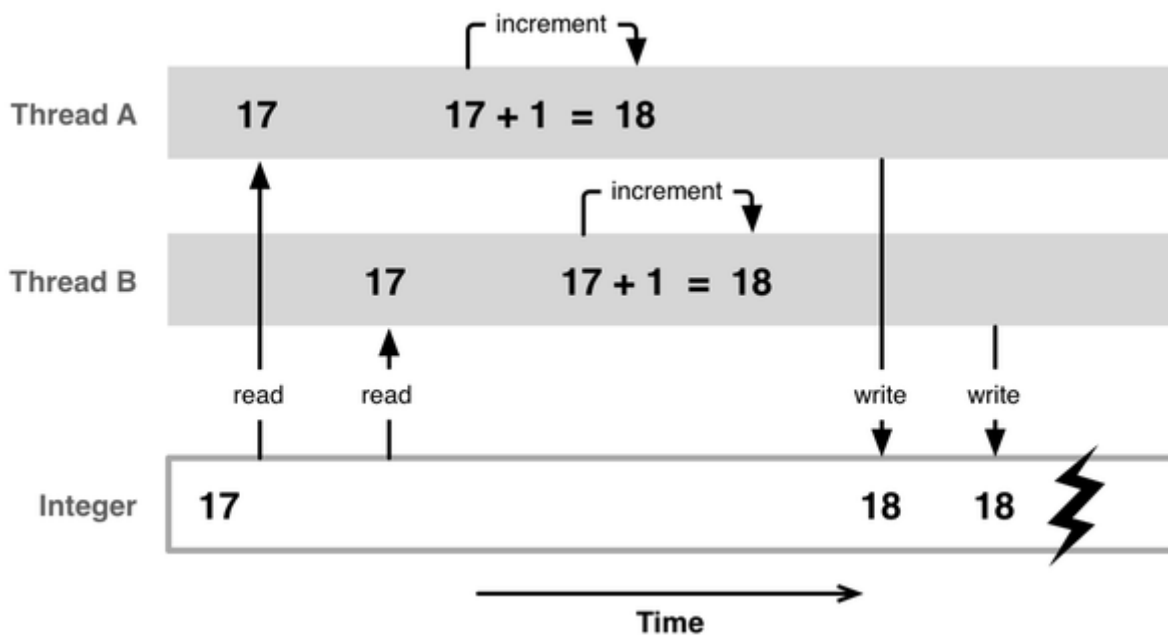
Here a class Rider creates an object of any class which implements the Bike interface.

```
1
2 class Rider
3 {
4     Bike b;
5     public void setBike(Bike b)
6     {
7         this.b = b;
8     }
9     void ride()
10    {
11        b.start();
12    }
```

Now the Spring Framework container can inject an object of any class that implements Bike interface, according to the requirement. This is how loose coupling works.

What do you mean by thread safe in Java? How does this (thread safe) mechanism work?

Java supports the concept of **concurrent**, or multi-threaded, programs. The basic unit of concurrency is a **thread**, and several threads can run within a single program. It's quite usual to have threads access a common data structure. In such access, it is possible that one thread interrupts another thread's *read* operation, to do its own *update* or *delete*. If not properly managed, these can lead to unexpected data contents!



It's clear from the picture, that both **Thread A** and **Thread B** both:

- read the integer into their memory
- increment its value, and then
- give it back into the data store.

These increment cycles are **not atomic**, because **Thread B** could access the data value before **Thread A** was done updating it. The programmer actually wanted two separate increments, to result in a final value of 19. What he got instead, was 18. Such a scenario, where the final state of data depends on the *relative* order of thread operations, is called a **race condition**. Race conditions affect the correctness of a program, and are very difficult to detect and debug.

A data structure is said to be **thread safe**, if operations can be done concurrently by many threads, *without the risk of race conditions*. When one thread is performing *read* on a particular data element, *no other* thread should be allowed to *modify* or *delete* this element (**atomicity**). In our earlier example, if the update cycles were **atomic**, the race condition would have been avoided. **Thread B** would have *waited* for **Thread A** to complete its increment, and *then* it would have gone ahead.

Java has certain **synchronization mechanisms** in place that allows for *data operations* to be **race-free**. The common ones are:

- `synchronized` method definitions (for these data operations)
- Lock-based access mechanisms, such as `ReentrantLock`
- Atomic data types, such as `AtomicInteger` and `LongAdder`
- thread-safe collection interfaces & their implementations, such as `ConcurrentMap` implemented by `ConcurrentHashMap`

Each of these is suited for different kinds of situations. Thread safety is well explained from a Java perspective in official documentation.

A point to note however, is that a pool of threads that perform *only a read operation* don't need exclusive access. Synchronization is not done here, and reads can be more efficient. Java thread-safety mechanisms put this fact to good use.

The underlying cost of thread safety tells us one thing: the concurrent programmer must choose between **efficiency** and **correctness**. And correctness wins, always.

What are design patterns used in Spring Framework?

- **Singleton Pattern:** Singleton-scoped beans
- **Factory Pattern:** Bean Factory classes
- **Prototype Pattern:** Prototype-scoped beans
- **Adapter Pattern:** Spring Web and Spring MVC
- **Proxy Pattern:** Spring Aspect Oriented Programming support
- **Template Method Pattern:** *JdbcTemplate*, *HibernateTemplate*, etc.
- **Front Controller:** Spring MVC *DispatcherServlet*
- **Data Access Object:** Spring DAO support
- **Model View Controller:** Spring MVC

What are different scopes of Spring Bean?

There are five scopes defined for Spring Beans.

1. **[singleton](#):** Only one instance of the bean will be created for each container. This is the default scope for the spring beans. While using this scope, make sure spring bean doesn't have shared instance variables otherwise it might lead to data inconsistency issues because it's not thread-safe.
2. **prototype:** A new instance will be created every time the bean is requested.
3. **request:** This is same as prototype scope, however it's meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
4. **session:** A new bean will be created for each HTTP session by the container.
5. **global-session:** This is used to create global session beans for Portlet applications.

Spring Framework is extendable and we can create our own scopes too, however most of the times we are good with the scopes provided by the framework.

To set spring bean scopes we can use "scope" attribute in bean element or `@Scope` annotation for annotation based configurations.

What is Spring IoC Container?

Inversion of Control (IoC) is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, the objects define their dependencies that are being injected by other assembler objects. Spring IoC container is the program that injects dependencies into an object and make it ready for our use.

Spring Framework IoC container classes are part of `org.springframework.beans` and `org.springframework.context` packages and provides us different ways to decouple the object dependencies.

Some of the useful `ApplicationContext` implementations that we use are;

- `AnnotationConfigApplicationContext`: For standalone java applications using annotations based configuration.
- `ClassPathXmlApplicationContext`: For standalone java applications using XML based configuration.
- `FileSystemXmlApplicationContext`: Similar to `ClassPathXmlApplicationContext` except that the xml configuration file can be loaded from anywhere in the file system.
- `AnnotationConfigWebApplicationContext` and `XmlWebApplicationContext` for web applications.

• What is a Spring Bean?

Any normal java class that is initialized by Spring IoC container is called Spring Bean. We use Spring `ApplicationContext` to get the Spring Bean instance.

Spring IoC container manages the life cycle of Spring Bean, bean scopes and injecting any required dependencies in the bean.

• What is the importance of Spring bean configuration file?

We use Spring Bean configuration file to define all the beans that will be initialized by Spring Context. When we create the instance of Spring `ApplicationContext`, it reads the spring bean xml file and initialize all of them. Once the context is initialized, we can use it to get different bean instances.

Apart from Spring Bean configuration, this file also contains spring MVC interceptors, view resolvers and other elements to support annotations based configurations.

What are different ways to configure a class as Spring Bean?

There are three different ways to configure Spring Bean.

1. **XML Configuration:** This is the most popular configuration and we can use bean element in context file to configure a Spring Bean. For example:
- 2.
3. `<bean name="myBean" class="com.journaldev.spring.beans.MyBean"></bean>`
4. **Java Based Configuration:** If you are using only annotations, you can configure a Spring bean using `@Bean` annotation. This annotation is used with `@Configuration` classes to configure a spring bean. Sample configuration is:

`@Configuration`

`@ComponentScan(value="com.journaldev.spring.main")`

```
public class MyConfiguration {
```

```
    @Bean
```

```
    public MyService getService(){
```

```
        return new MyService();
```

```
    }
```

```
}
```

- To get this bean from spring context, we need to use following code snippet:

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

```
MyService service = ctx.getBean(MyService.class);
```

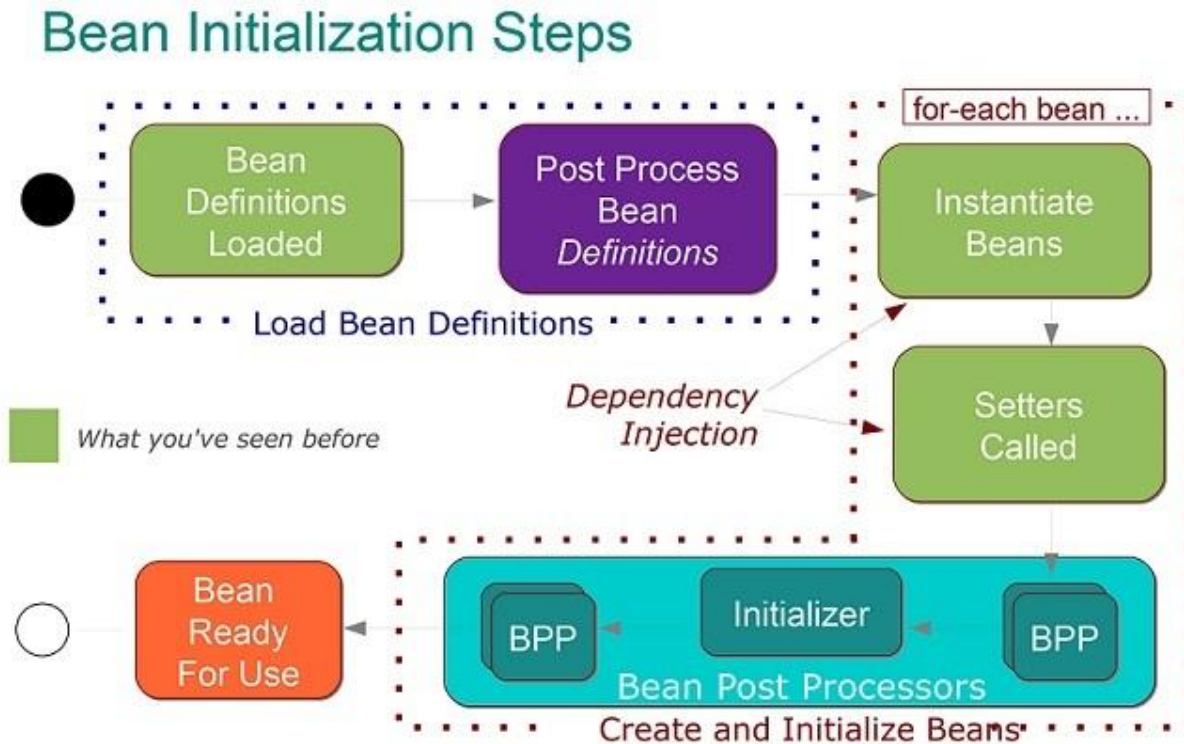
- **Annotation Based Configuration:** We can also use `@Component`, `@Service`, `@Repository` and `@Controller` annotations with classes to configure them to be as spring bean. For these, we would need to provide base package location to scan for these classes. For example:

```
<context:component-scan base-package="com.journaldev.spring" />
```

• What is Spring Bean life cycle?

Spring Beans are initialized by Spring Container and all the dependencies are also injected. When context is destroyed, it also destroys all the initialized beans. This works well in most of the cases but sometimes we want to initialize other resources or do some validation before making our beans ready to use. Spring framework provides support for post-initialization and pre-destroy methods in spring beans.

We can do this by two ways – by implementing `InitializingBean` and `DisposableBean` interfaces or using **init-method** and **destroy-method** attribute in spring bean configurations. For more details, please read [Spring Bean Life Cycle Methods](#).



• How to get ServletContext and ServletConfig object in a Spring Bean?

There are two ways to get Container specific objects in the spring bean.

1. Implementing Spring *Aware interfaces, for these `ServletContextAware` and `ServletConfigAware` interfaces, for complete example of these aware interfaces, please read [Spring Aware Interfaces](#)
2. Using `@Autowired` annotation with bean variable of type `ServletContext` and `ServletConfig`. They will work only in servlet container specific environment only though.
- 3.
4. `@Autowired`
5. `ServletContext servletContext;`

How can we inject beans in Spring?

A few different options exist:

- Setter Injection
- Constructor Injection
- Field Injection

The configuration can be done using XML files or annotations.

Which is the best way of injecting beans and why?

The recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. Constructor injection allows injecting values to immutable fields and makes testing easier.

Q7. What is the difference between *BeanFactory* and *ApplicationContext*?

BeanFactory is an interface representing a container that provides and manages bean instances. The default implementation instantiates beans lazily when *getBean()* is called.

ApplicationContext is an interface representing a container holding all information, metadata, and beans in the application. It also extends the *BeanFactory* interface but the default implementation instantiates beans eagerly when the application starts. This behavior can be overridden for individual beans.

Are singleton beans thread-safe?

No, singleton beans are not thread-safe, as thread safety is about execution, whereas the singleton is a design pattern focusing on creation. Thread safety depends only on the bean implementation itself.

Q12. What does the Spring bean lifecycle look like?

First, a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some initialization to get it into a usable state. After that, when the bean is no longer required, it will be removed from the IoC container.

• What is Bean wiring and @Autowired annotation?

The process of injection spring bean dependencies while initializing it called Spring Bean Wiring.

Usually it's best practice to do the explicit wiring of all the bean dependencies, but spring framework also supports autowiring. We can use `@Autowired` annotation with fields or methods for **autowiring byType**. For this annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by **context:annotation-config** element.

For more details about `@Autowired` annotation, please read [Spring Autowire Example](#).

• What are different types of Spring Bean autowiring?

There are four types of autowiring in Spring framework.

1. **autowire byName**
2. **autowire byType**
3. **autowire by constructor**
4. autowiring by **@Autowired** and **@Qualifier** annotations

Prior to Spring 3.1, **autowire by autodetect** was also supported that was similar to autowire by constructor or byType. For more details about these options, please read [Spring Bean Autowiring](#).

• Does Spring Bean provide thread safety?

The default scope of Spring bean is singleton, so there will be only one instance per context. That means that all the having a class level variable that any thread can update will lead to inconsistent data. Hence in default mode spring beans are not thread-safe.

However we can change spring bean scope to request, prototype or session to achieve thread-safety at the cost of performance. It's a design decision and based on the project requirements.

What are some of the important features and advantages of Spring Framework?

Spring targets to make Java EE development easier. Here are the advantages of using it:

- **Lightweight:** there is a slight overhead of using the framework in development
- **Inversion of Control (IoC):** Spring container takes care of wiring dependencies of various objects, instead of creating or looking for dependent objects
- **Aspect Oriented Programming (AOP):** Spring supports AOP to separate business logic from system services
- **IoC container:** it manages Spring Bean life cycle and project specific configurations
- **MVC framework:** that is used to create web applications or RESTful web services, capable of returning XML/JSON responses
- **Transaction management:** reduces the amount of boiler-plate code in JDBC operations, file uploading, etc., either by using Java annotations or by Spring Bean XML configuration file
- **Exception Handling:** Spring provides a convenient API for translating technology-specific exceptions into unchecked exceptions

What are the benefits of using Spring?

Some of the advantages of using Spring Framework are:

- Reducing direct dependencies between different components of the application, usually Spring IoC container is responsible for initializing resources or beans and inject them as dependencies.
- Writing unit test cases are easy in Spring framework because our business logic doesn't have direct dependencies with actual resource implementation classes. We can easily write a test configuration and inject our mock beans for testing purposes.
- Reduces the amount of boiler-plate code, such as initializing objects, open/close resources. I like JdbcTemplate class a lot because it helps us in removing all the boiler-plate code that comes with JDBC programming.
- Spring framework is divided into several modules, it helps us in keeping our application lightweight. For example, if we don't need Spring transaction management features, we don't need to add that dependency in our project.
- Spring framework support most of the Java EE features and even much more. It's always on top of the new technologies, for example there is a Spring project for Android to help us write better code for native android applications. This makes spring framework a complete package and we don't need to look after different framework for different requirements.

What Spring sub-projects do you know? Describe them briefly.

- **Core** – a key module that provides fundamental parts of the framework, like IoC or DI
- **JDBC** – this module enables a JDBC-abstraction layer that removes the need to do JDBC coding for specific vendor databases
- **ORM integration** – provides integration layers for popular object-relational mapping APIs, such as JPA, JDO, and Hibernate
- **Web** – a web-oriented integration module, providing multipart file upload, Servlet listeners, and web-oriented application context functionalities
- **MVC framework** – a web module implementing the Model View Controller design pattern
- **AOP module** – aspect-oriented programming implementation allowing the definition of clean method-interceptors and pointcuts

What do you understand by Aspect Oriented Programming?

Enterprise applications have some common cross-cutting concerns that is applicable for different types of Objects and application modules, such as logging, transaction management, data validation, authentication etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in AOP application modularity is achieved by Aspects and they are configured to cut across different classes methods.

AOP takes out the direct dependency of cross-cutting tasks from classes that is not possible in normal object oriented programming. For example, we can have a separate class for logging but again the classes will have to call these methods for logging the data.

What is Aspect, Advice, Pointcut, JointPoint and Advice Arguments in AOP?

Aspect: Aspect is a class that implements cross-cutting concerns, such as transaction management. Aspects can be a normal class configured and then configured in Spring Bean configuration file or we can use Spring AspectJ support to declare a class as Aspect using `@Aspect` annotation.

Advice: Advice is the action taken for a particular join point. In terms of programming, they are methods that gets executed when a specific join point with matching pointcut is reached in the application. You can think of Advices as [Spring interceptors](#) or [Servlet Filters](#).

Pointcut: Pointcut are regular expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points. Spring framework uses the AspectJ pointcut expression language for determining the join points where advice methods will be applied.

Join Point: A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.

Advice Arguments: We can pass arguments in the advice methods. We can use `args()` expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where argument type is determined.

• What is the difference between Spring AOP and AspectJ AOP?

AspectJ is the industry-standard implementation for Aspect Oriented Programming whereas Spring implements AOP for some cases. Main differences between Spring AOP and AspectJ are:

- Spring AOP is simpler to use than AspectJ because we don't need to worry about the weaving process.
- Spring AOP supports AspectJ annotations, so if you are familiar with AspectJ then working with Spring AOP is easier.
- Spring AOP supports only proxy-based AOP, so it can be applied only to method execution join points. AspectJ support all kinds of pointcuts.
- One of the shortcoming of Spring AOP is that it can be applied only to the beans created through Spring Context.

• What is ViewResolver in Spring?

`ViewResolver` implementations are used to resolve the view pages by name. Usually we configure it in the spring bean configuration file. For example:

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <beans:property name="prefix" value="/WEB-INF/views/" />
    <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

`InternalResourceViewResolver` is one of the implementation of `ViewResolver` interface and we are providing the view pages directory and suffix location through the bean properties. So if a controller handler method returns “home”, view resolver will use view page located at `/WEB-INF/views/home.jsp`.

• What is a `MultipartResolver` and when its used?

`MultipartResolver` interface is used for uploading files – `CommonsMultipartResolver` and `StandardServletMultipartResolver` are two implementations provided by spring framework for file uploading. By default there are no multipart resolvers configured but to use them for uploading files, all we need to define a bean named “`multipartResolver`” with type as `MultipartResolver` in spring bean configurations.

Once configured, any multipart request will be resolved by the configured `MultipartResolver` and pass on a wrapped `HttpServletRequest`. Then it’s used in the controller class to get the file and process it. For a complete example, please read [Spring MVC File Upload Example](#).

• How to handle exceptions in Spring MVC Framework?

Spring MVC Framework provides following ways to help us achieving robust exception handling.

1. **Controller Based** – We can define exception handler methods in our controller classes. All we need is to annotate these methods with `@ExceptionHandler` annotation.
2. **Global Exception Handler** – Exception Handling is a cross-cutting concern and Spring provides `@ControllerAdvice` annotation that we can use with any class to define our global exception handler.
3. **HandlerExceptionResolver implementation** – For generic exceptions, most of the times we serve static pages. Spring Framework provides `HandlerExceptionResolver` interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

For a complete example, please read [Spring Exception Handling Example](#).

• How to create `ApplicationContext` in a Java Program?

There are following ways to create spring context in a standalone java program.

1. **AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
2. **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.
3. **FileSystemXmlApplicationContext**: This is similar to `ClassPathXmlApplicationContext` except that the xml configuration file can be loaded from anywhere in the file system.

• Can we have multiple Spring configuration files?

For Spring MVC applications, we can define multiple spring context configuration files through `contextConfigLocation`. This location string can consist of multiple locations separated by any number of commas and spaces. For example;

```
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/appServlet/servlet-context.xml,/WEB-
INF/spring/appServlet/servlet-jdbc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

We can also define multiple root level spring configurations and load it through context-param. For example;

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml /WEB-INF/spring/root-
security.xml</param-value>
</context-param>
```

Another option is to use import element in the context configuration file to import other configurations, for example:

```
<beans:import resource="spring-jdbc.xml"/>
```

• What is ContextLoaderListener?

ContextLoaderListener is the listener class used to load root context and define spring bean configurations that will be visible to all other contexts. It's configured in web.xml file as:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>

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

• What are the minimum configurations needed to create Spring MVC application?

For creating a simple Spring MVC application, we would need to do following tasks.

- Add `spring-context` and `spring-webmvc` dependencies in the project.
- Configure `DispatcherServlet` in the `web.xml` file to handle requests through spring container.
- Spring bean configuration file to define beans, if using annotations then it has to be configured here. Also we need to configure view resolver for view pages.
- Controller class with request mappings defined to handle the client requests.

Above steps should be enough to create a simple Spring MVC Hello World application.

• How would you relate Spring MVC Framework to MVC architecture?

As the name suggests Spring MVC is built on top of **Model-View-Controller** architecture. `DispatcherServlet` is the Front Controller in the Spring MVC application that takes care of all the incoming requests and delegate it to different controller handler methods.

Model can be any Java Bean in the Spring Framework, just like any other MVC framework Spring provides automatic binding of form data to java beans. We can set model beans as attributes to be used in the view pages.

View Pages can be JSP, static HTMLs etc. and view resolvers are responsible for finding the correct view page. Once the view page is identified, control is given back to the `DispatcherServlet` controller. `DispatcherServlet` is responsible for rendering the view and returning the final response to the client.

• How to achieve localization in Spring MVC applications?

Spring provides excellent support for localization or i18n through resource bundles. Basis steps needed to make our application localized are:

1. Creating message resource bundles for different locales, such as messages_en.properties, messages_fr.properties etc.
2. Defining messageSource bean in the spring bean configuration file of type ResourceBundleMessageSource or ReloadableResourceBundleMessageSource.
3. For change of locale support, define localeResolver bean of type CookieLocaleResolver and configure LocaleChangeInterceptor interceptor. Example configuration can be like below:
- 4.
5. `<beans:bean id="messageSource"`
6. `class="org.springframework.context.support.ReloadableResourceBundleMessageSource">`
7. `<beans:property name="basename" value="classpath:messages" />`
8. `<beans:property name="defaultEncoding" value="UTF-8" />`
9. `</beans:bean>`
- 10.
11. `<beans:bean id="localeResolver"`
12. `class="org.springframework.web.servlet.i18n.CookieLocaleResolver">`
13. `<beans:property name="defaultLocale" value="en" />`
14. `<beans:property name="cookieName" value="myAppLocaleCookie"></beans:property>`
15. `<beans:property name="cookieMaxAge" value="3600"></beans:property>`
16. `</beans:bean>`
- 17.
18. `<interceptors>`
19. `<beans:bean`
20. `class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">`
21. `<beans:property name="paramName" value="locale" />`
22. `</beans:bean>`
23. `</interceptors>`
23. Use spring:message element in the view pages with key names, DispatcherServlet picks the corresponding value and renders the page in corresponding locale and return as response.

For a complete example, please read [Spring Localization Example](#).

• How can we use Spring to create Restful Web Service returning JSON response?

We can use Spring Framework to create Restful web services that returns JSON data. Spring provides integration with [Jackson JSON](#) API that we can use to send JSON response in restful web service.

We would need to do following steps to configure our Spring MVC application to send JSON response:

1. Adding [Jackson](#) JSON dependencies, if you are using Maven it can be done with following code:
- 2.
3. `<!-- Jackson -->`
4. `<dependency>`
5. `<groupId>com.fasterxml.jackson.core</groupId>`
6. `<artifactId>jackson-databind</artifactId>`
7. `<version>${jackson.databind-version}</version>`
8. `</dependency>`
9. Configure RequestMappingHandlerAdapter bean in the spring bean configuration file and set the messageConverters property to MappingJackson2HttpMessageConverter bean. Sample configuration will be:
- 10.
11. `<!-- Configure to plugin JSON as request and response in method handler -->`
12. `<beans:bean`
13. `class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">`
14. `<beans:property name="messageConverters">`
15. `<beans:list>`
16. `<beans:ref bean="jsonMessageConverter"/>`
17. `</beans:list>`
18. `</beans:property>`
19. `</beans:bean>`
20. `<!-- Configure bean to convert JSON to POJO and vice versa -->`
21. `<beans:bean id="jsonMessageConverter"`
22. `class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">`
23. `</beans:bean>`
23. In the controller handler methods, return the Object as response using @ResponseBody annotation. Sample code:
- 24.
25. `@RequestMapping(value = EmpRestURIConstants.GET_EMP, method = RequestMethod.GET)`

```

26. public @ResponseBody Employee getEmployee(@PathVariable("id") int empId) {
27.     logger.info("Start getEmployee. ID="+empId);
28.
29.     return empData.get(empId);
30. }

```

31. You can invoke the rest service through any API, but if you want to use Spring then we can easily do it using `RestTemplate` class.

For a complete example, please read [Spring Restful Webservice Example](#).

• What are some of the important Spring annotations you have used?

Some of the Spring annotations that I have used in my project are:

- **@Controller** – for controller classes in Spring MVC project.
- **@RequestMapping** – for configuring URI mapping in controller handler methods. This is a very important annotation, so you should go through [Spring MVC RequestMapping Annotation Examples](#)
- **@ResponseBody** – for sending Object as response, usually for sending XML or JSON data as response.
- **@PathVariable** – for mapping dynamic values from the URI to handler method arguments.
- **@Autowired** – for autowiring dependencies in spring beans.
- **@Qualifier** – with **@Autowired** annotation to avoid confusion when multiple instances of bean type is present.
- **@Service** – for service classes.
- **@Scope** – for configuring scope of the spring bean.
- **@Configuration**, **@ComponentScan** and **@Bean** – for java based configurations.
- AspectJ annotations for configuring aspects and advices, **@Aspect**, **@Before**, **@After**, **@Around**, **@Pointcut** etc.

• Can we send an Object as the response of Controller handler method?

Yes we can, using **@ResponseBody** annotation. This is how we send JSON or XML based response in restful web services.

• How to upload file in Spring MVC Application?

Spring provides built-in support for uploading files through **MultipartResolver** interface implementations. It's very easy to use and requires only configuration changes to get it working. Obviously we would need to write controller handler method to handle the incoming file and process it. For a complete example, please refer [Spring File Upload Example](#).

• How to validate form data in Spring Web MVC Framework?

Spring supports JSR-303 annotation based validations as well as provide Validator interface that we can implement to create our own custom validator. For using JSR-303 based validation, we need to annotate bean variables with the required validations.

For custom validator implementation, we need to configure it in the controller class. For a complete example, please read [Spring MVC Form Validation Example](#).

• What is Spring MVC Interceptor and how to use it?

Spring MVC Interceptors are like Servlet Filters and allow us to intercept client request and process it. We can intercept client request at three places – **preHandle**, **postHandle** and **afterCompletion**.

We can create spring interceptor by implementing `HandlerInterceptor` interface or by extending abstract class **HandlerInterceptorAdapter**.

We need to configure interceptors in the spring bean configuration file. We can define an interceptor to intercept all the client requests or we can configure it for specific URI mapping too. For a detailed example, please refer [Spring MVC Interceptor Example](#).

- **What is Spring JdbcTemplate class and how to use it?**

Spring Framework provides excellent integration with JDBC API and provides JdbcTemplate utility class that we can use to avoid boiler-plate code from our database operations logic such as Opening/Closing Connection, ResultSet, PreparedStatement etc.

For JdbcTemplate example, please refer [Spring JDBC Example](#).

- **How to use Tomcat JNDI DataSource in Spring Web Application?**

For using servlet container configured JNDI DataSource, we need to configure it in the spring bean configuration file and then inject it to spring beans as dependencies. Then we can use it with JdbcTemplate to perform database operations.

Sample configuration would be:

```
<beans:bean id="dbDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <beans:property name="jndiName" value="java:comp/env/jdbc/MyLocalDB"/>
</beans:bean>
```

For complete example, please refer [Spring Tomcat JNDI Example](#).

- **How would you achieve Transaction Management in Spring?**

Spring framework provides transaction management support through Declarative Transaction Management as well as programmatic transaction management. Declarative transaction management is most widely used because it's easy to use and works in most of the cases.

We use annotate a method with @Transactional annotation for Declarative transaction management. We need to configure transaction manager for the DataSource in the spring bean configuration file.

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

- **What is Spring DAO?**

Spring DAO support is provided to work with data access technologies like JDBC, Hibernate in a consistent and easy way. For example we have JdbcDaoSupport, HibernateDaoSupport, JdoDaoSupport and JpaDaoSupport for respective technologies.

Spring DAO also provides consistency in exception hierarchy and we don't need to catch specific exceptions.

- **How to integrate Spring and Hibernate Frameworks?**

We can use Spring ORM module to integrate Spring and Hibernate frameworks, if you are using Hibernate 3+ where SessionFactory provides current session, then you should avoid using HibernateTemplate or HibernateDaoSupport classes and better to use DAO pattern with dependency injection for the integration.

Also Spring ORM provides support for using Spring declarative transaction management, so you should utilize that rather than going for hibernate boiler-plate code for transaction management.

For better understanding you should go through following tutorials:

- [Spring Hibernate Integration Example](#)
- [Spring MVC Hibernate Integration Example](#)

• What is Spring Security?

Spring security framework focuses on providing both authentication and authorization in java applications. It also takes care of most of the common security vulnerabilities such as CSRF attack.

It's very beneficial and easy to use Spring security in web applications, through the use of annotations such as `@EnableWebSecurity`. You should go through following posts to learn how to use Spring Security framework.

- [Spring Security in Servlet Web Application](#)
- [Spring MVC and Spring Security Integration Example](#)

• How to inject a java.util.Properties into a Spring Bean?

We need to define propertyConfigurer bean that will load the properties from the given property file. Then we can use Spring EL support to inject properties into other bean dependencies. For example;

```
<bean id="propertyConfigurer"
      class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
    <property name="location" value="/WEB-INF/application.properties" />
</bean>

<bean class="com.journaldev.spring.EmployeeDaoImpl">
    <property name="maxReadResults" value="${results.read.max}" />
</bean>
```

If you are using annotation to configure the spring bean, then you can inject property like below.

```
@Value("${maxReadResults}")
private int maxReadResults;
```

• Name some of the design patterns used in Spring Framework?

Spring Framework is using a lot of design patterns, some of the common ones are:

1. Singleton Pattern: Creating beans with default scope.
2. [Factory Pattern](#): Bean Factory classes
3. [Prototype Pattern](#): Bean scopes
4. [Adapter Pattern](#): Spring Web and Spring MVC
5. [Proxy Pattern](#): Spring Aspect Oriented Programming support
6. [Template Method Pattern](#): JdbcTemplate, HibernateTemplate etc
7. Front Controller: Spring MVC DispatcherServlet
8. Data Access Object: Spring DAO support
9. Dependency Injection and Aspect Oriented Programming

• What are some of the best practices for Spring Framework?

Some of the best practices for Spring Framework are:

1. Avoid version numbers in schema reference, to make sure we have the latest configs.
2. Divide spring bean configurations based on their concerns such as spring-jdbc.xml, spring-security.xml.
3. For spring beans that are used in multiple contexts in Spring MVC, create them in the root context and initialize with listener.
4. Configure bean dependencies as much as possible, try to avoid autowiring as much as possible.
5. For application level properties, best approach is to create a property file and read it in the spring bean configuration file.
6. For smaller applications, annotations are useful but for larger applications annotations can become a pain. If we have all the configuration in xml files, maintaining it will be easier.

7. Use correct annotations for components for understanding the purpose easily. For services use `@Service` and for DAO beans use `@Repository`.
8. Spring framework has a lot of modules, use what you need. Remove all the extra dependencies that gets usually added when you create projects through Spring Tool Suite templates.
9. If you are using Aspects, make sure to keep the join point as narrow as possible to avoid advice on unwanted methods. Consider custom annotations that are easier to use and avoid any issues.
10. Use dependency injection when there is actual benefit, just for the sake of loose-coupling don't use it because it's harder to maintain.

Spring Framework Annotations

The Java Programming language provided support for Annotations from Java 5.0. Leading Java frameworks were quick to adopt annotations and the Spring Framework started using annotations from the release 2.5. Due to the way they are defined, annotations provide a lot of context in their declaration.

Prior to annotations, the behavior of the Spring Framework was largely controlled through XML configuration. Today, the use of annotations provide us tremendous capabilities in how we configure the behaviours of the Spring Framework.

In this post, we'll take a look at the annotations available in the Spring Framework.

Core Spring Framework Annotations

@Required

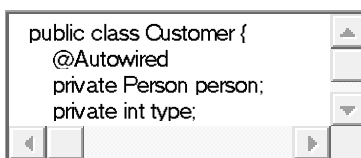
This annotation is applied on bean setter methods. Consider a scenario where you need to enforce a required property. The `@Required` annotation indicates that the affected bean must be populated at configuration time with the required property. Otherwise an exception of type `BeanInitializationException` is thrown.

@Autowired

This annotation is applied on fields, setter methods, and constructors. The `@Autowired` annotation injects object dependency implicitly.

When you use `@Autowired` on fields and pass the values for the fields using the property name, Spring will automatically assign the fields with the passed values.

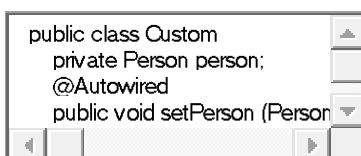
You can even use `@Autowired` on private properties, as shown below. (This is a very poor practice though!)



```
public class Customer {
    @Autowired
    private Person person;
    private int type;
}
```

```
1 public class Customer {
2     @Autowired
3     private Person person;
4     private int type;
5 }
```

When you use `@Autowired` on setter methods, Spring tries to perform the by Type autowiring on the method. You are instructing Spring that it should initiate this property using setter method where you can add your custom code, like initializing any other property with this property.



```
public class Customer {
    private Person person;
    @Autowired
    public void setPerson (Person person) {
        // Custom code here
    }
}
```

```
1 public class Customer {
2     private Person person;
3     @Autowired
```

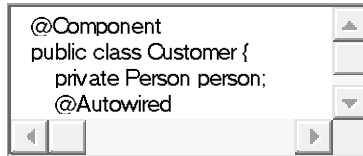
```

4  public void setPerson (Person person) {
5      this.person=person;
6  }
7  }

```

Consider a scenario where you need instance of class A, but you do not store A in the field of the class. You just use A to obtain instance of B, and you are storing B in this field. In this case setter method autowiring will better suite you. You will not have class level unused fields.

When you use `@Autowired` on a constructor, constructor injection happens at the time of object creation. It indicates the constructor to autowire when used as a bean. One thing to note here is that only one constructor of any bean class can carry the `@Autowired` annotation.



```

@Component
public class Customer {
    private Person person;
    @Autowired

```

```

1 @Component
2 public class Customer {
3     private Person person;
4     @Autowired
5     public Customer (Person person) {
6         this.person=person;
7     }
8 }

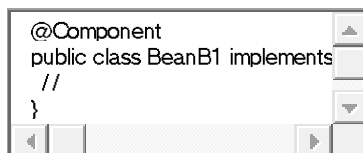
```

NOTE: As of Spring 4.3, `@Autowired` became optional on classes with a single constructor. In the above example, Spring would still inject an instance of the `Person` class if you omitted the `@Autowired` annotation.

@Qualifier

This annotation is used along with `@Autowired` annotation. When you need more control of the dependency injection process, `@Qualifier` can be used. `@Qualifier` can be specified on individual constructor arguments or method parameters. This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property.

Consider an example where an interface `BeanInterface` is implemented by two beans `BeanB1` and `BeanB2`.



```

@Component
public class BeanB1 implements BeanInterface {
    //
}
@Component
public class BeanB2 implements BeanInterface {
    //
}

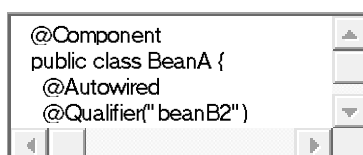
```

```

1 @Component
2 public class BeanB1 implements BeanInterface {
3     //
4 }
5 @Component
6 public class BeanB2 implements BeanInterface {
7     //
8 }

```

Now if `BeanA` autowires this interface, Spring will not know which one of the two implementations to inject. One solution to this problem is the use of the `@Qualifier` annotation.



```

@Component
public class BeanA {
    @Autowired
    @Qualifier("beanB2")

```

```

1 @Component
2 public class BeanA {
3     @Autowired
4     @Qualifier("beanB2")

```

```

5 private BeanInterface dependency;
6 ...
7 }

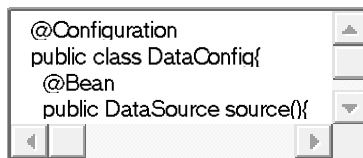
```

With the `@Qualifier` annotation added, Spring will now know which bean to autowire where `beanB2` is the name of `BeanB2`. Learn the Spring Framework with my Spring Framework 5 – Beginner to Guru Course!

@Configuration

This annotation is used on classes which define beans. `@Configuration` is an analog for XML configuration file – it is configuration using Java class. Java class annotated with `@Configuration` is a configuration by itself and will have methods to instantiate and configure the dependencies.

Here is an example:



```

1 @Configuration
2 public class DataConfig{
3     @Bean
4     public DataSource source(){
5         DataSource source = new OracleDataSource();
6         source.setURL();
7         source.setUser();
8         return source;
9     }
10    @Bean
11    public PlatformTransactionManager manager(){
12        PlatformTransactionManager manager = new BasicDataSourceTransactionManager();
13        manager.setDataSource(source());
14        return manager;
15    }
16 }

```

@ComponentScan

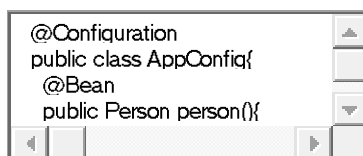
This annotation is used with `@Configuration` annotation to allow Spring to know the packages to scan for annotated components. `@ComponentScan` is also used to specify base packages using `basePackageClasses` or `basePackage` attributes to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

Checkout this [post](#) for an in depth look at the Component Scan annotation.

@Bean

This annotation is used at the method level. `@Bean` annotation works with `@Configuration` to create Spring beans. As mentioned earlier, `@Configuration` will have methods to instantiate and configure dependencies. Such methods will be annotated with `@Bean`. The method annotated with this annotation works as bean ID and it creates and returns the actual bean.

Here is an example:



```

1 @Configuration
2 public class AppConfig{
3     @Bean
4     public Person person(){
5         return new Person(address());
6     }
7     @Bean

```

```
8 public Address address(){
9     return new Address();
10 }
11 }
```

@Lazy

This annotation is used on component classes. By default all autowired dependencies are created and configured at startup. But if you want to initialize a bean lazily, you can use `@Lazy` annotation over the class. This means that the bean will be created and initialized only when it is first requested for. You can also use this annotation on `@Configuration` classes. This indicates that all `@Bean` methods within that `@Configuration` should be lazily initialized.

@Value

This annotation is used at the field, constructor parameter, and method parameter level. The `@Value` annotation indicates a default value expression for the field or parameter to initialize the property with. As the `@Autowired` annotation tells Spring to inject object into another when it loads your application context, you can also use `@Value` annotation to inject values from a property file into a bean's attribute. It supports both `#{...}` and `${...}` placeholders.

Spring Framework Stereotype Annotations

@Component

This annotation is used on classes to indicate a Spring component. The `@Component` annotation marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context.

@Controller

The `@Controller` annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

@Service

This annotation is used on a class. The `@Service` marks a Java class that performs some service, such as execute business logic, perform calculations and call external APIs. This annotation is a specialized form of the `@Component` annotation intended to be used in the service layer.

@Repository

This annotation is used on Java classes which directly access the database. The `@Repository` annotation works as marker for any class that fulfills the role of repository or Data Access Object.

This annotation has a automatic translation feature. For example, when an exception occurs in the `@Repository` there is a handler for that exception and there is no need to add a try catch block.

Spring Boot Annotations

@EnableAutoConfiguration

This annotation is usually placed on the main application class. The `@EnableAutoConfiguration` annotation implicitly defines a base "search package". This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

@SpringBootApplication

This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the `@SpringBootApplication` must be kept in the base package. The one thing that the `@SpringBootApplication` does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with `@SpringBootApplication` in `com.example` then `@SpringBootApplication` will scan all its sub-packages, such as `com.example.a`, `com.example.b`, and `com.example.a.x`.

The `@SpringBootApplication` is a convenient annotation that adds all the following:

- `@Configuration`
- `@EnableAutoConfiguration`
- `@ComponentScan`

Spring MVC and REST Annotations

@Controller

This annotation is used on Java classes that play the role of controller in your application. The @Controller annotation allows autodetection of component classes in the classpath and auto-registering bean definitions for them. To enable autodetection of such annotated controllers, you can add component scanning to your configuration. The Java class annotated with @Controller is capable of handling multiple request mappings.

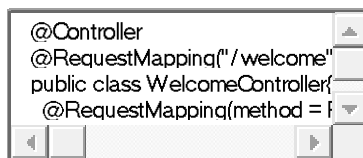
This annotation can be used with Spring MVC and Spring WebFlux.

@RequestMapping

This annotation is used both at class and method level. The @RequestMapping annotation is used to map web requests onto specific handler classes and handler methods. When @RequestMapping is used on class level it creates a base URI for which the controller will be used. When this annotation is used on methods it will give you the URI on which the handler methods will be executed. From this you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.

Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the method attribute of @RequestMapping with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Here is a basic example on how a controller along with request mappings work:



```
1 @Controller
2 @RequestMapping("/welcome")
3 public class WelcomeController{
4     @RequestMapping(method = RequestMethod.GET)
5     public String welcomeAll(){
6         return "welcome all";
7     }
8 }
```

In this example only GET requests to /welcome is handled by the welcomeAll() method.

This annotation also can be used with Spring MVC and Spring WebFlux.

The @RequestMapping annotation is very versatile. Please see my in depth post on [Request Mapping here](#).

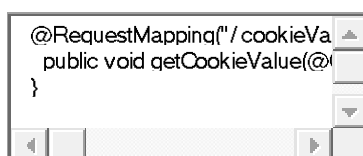
@CookieValue

This annotation is used at method parameter level. @CookieValue is used as argument of request mapping method. The HTTP cookie is bound to the @CookieValue parameter for a given cookie name. This annotation is used in the method annotated with @RequestMapping.

Let us consider that the following cookie value is received with a http request:

JSESSIONID=418AB76CD83EF94U85YD34W

To get the value of the cookie, use @CookieValue like this:



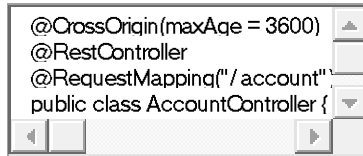
```
1 @RequestMapping("/cookieValue")
2 public void getCookieValue(@CookieValue("JSESSIONID") String cookie){
3 }
```

@CrossOrigin

This annotation is used both at class and method level to enable cross origin requests. In many cases the host that serves JavaScript will be different from the host that serves the data. In such a case Cross Origin Resource Sharing (CORS) enables cross-domain communication. To enable this communication you just need to add the @CrossOrigin annotation.

By default the @CrossOrigin annotation allows all origin, all headers, the HTTP methods specified in the @RequestMapping annotation and maxAge of 30 min. You can customize the behavior by specifying the corresponding attribute values.

An example to use @CrossOrigin at both controller and handler method levels is this.



```
1  @CrossOrigin(maxAge = 3600)
2  @RestController
3  @RequestMapping("/account")
4  public class AccountController {
5
6      @CrossOrigin(origins = "http://example.com")
7      @RequestMapping("/message")
8      public Message getMessage() {
9          // ...
10     }
11     @RequestMapping("/note")
12     public Note getNote() {
13         // ...
14     }
15 }
16 }
```

In this example, both getMessage() and getNote() methods will have a maxAge of 3600 seconds. Also, getMessage() will only allow cross-origin requests from http://example.com, while getNote() will allow cross-origin requests from all hosts.

Composed @RequestMapping Variants

Spring framework 4.3 introduced the following method-level variants of @RequestMapping annotation to better express the semantics of the annotated methods. Using these annotations have become the standard ways of defining the endpoints. They act as wrapper to @RequestMapping.

These annotations can be used with Spring MVC and Spring WebFlux.

@GetMapping

This annotation is used for mapping HTTP GET requests onto specific handler methods. @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET)

@PostMapping

This annotation is used for mapping HTTP POST requests onto specific handler methods. @PostMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST)

@PutMapping

This annotation is used for mapping HTTP PUT requests onto specific handler methods. @PutMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PUT)

@PatchMapping

This annotation is used for mapping HTTP PATCH requests onto specific handler methods. @PatchMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PATCH)

@DeleteMapping

This annotation is used for mapping HTTP DELETE requests onto specific handler methods. @DeleteMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.DELETE)

@ExceptionHandler

This annotation is used at method levels to handle exception at the controller level. The @ExceptionHandler annotation is used to define the class of exception it will catch. You can use this annotation on methods that should be invoked to handle an exception. The @ExceptionHandler values can be set to an array of Exception types. If an exception is thrown that matches one of the types in the list, then the method annotated with matching @ExceptionHandler will be invoked.

@InitBinder

This annotation is a method level annotation that plays the role of identifying the methods which initialize the WebDataBinder - a DataBinder that binds the request parameter to JavaBean objects. To customise request parameter data binding , you can use @InitBinder annotated methods within our controller. The methods annotated with @InitBinder all argument types that handler methods support. The @InitBinder annotated methods will get called for each HTTP request if you don't specify the value element of this annotation. The value element can be a single or multiple form names or request parameters that the init binder method is applied to.

@Mappings and @Mapping

This annotation is used on fields. The @Mapping annotation is a meta annotation that indicates a web mapping annotation. When mapping different field names, you need to configure the source field to its target field and to do that you have to add the @Mappings annotation. This annotation accepts an array of @Mapping having the source and the target fields.

@MatrixVariable

This annotation is used to annotate request handler method arguments so that Spring can inject the relevant bits of matrix URI. Matrix variables can appear on any segment each separated by a semicolon. If a URL contains matrix variables, the request mapping pattern must represent them with a URI template. The @MatrixVariable annotation ensures that the request is matched with the correct matrix variables of the URI.

@PathVariable

This annotation is used to annotate request handler method arguments. The @RequestMapping annotation can be used to handle dynamic changes in the URI where certain URI value acts as a parameter. You can specify this parameter using a regular expression. The @PathVariable annotation can be used declare this parameter.

@RequestAttribute

This annotation is used to bind the request attribute to a handler method parameter. Spring retrieves the named attributes value to populate the parameter annotated with @RequestAttribute. While the @RequestParam annotation is used bind the parameter values from query string, the @RequestAttribute is used to access the objects which have been populated on the server side.

@RequestBody

This annotation is used to annotate request handler method arguments. The @RequestBody annotation indicates that a method parameter should be bound to the value of the HTTP request body. The HttpMessageConveter is responsible for converting from the HTTP request message to object.

@RequestHeader

This annotation is used to annotate request handler method arguments. The @RequestHeader annotation is used to map controller parameter to request header value. When Spring maps the request, @RequestHeader checks the header with the name specified within the annotation and binds its value to the handler method parameter. This annotation helps you to get the header details within the controller class.

@RequestParam

This annotation is used to annotate request handler method arguments. Sometimes you get the parameters in the request URL, mostly in GET requests. In that case, along with the @RequestMapping annotation you can use the @RequestParam annotation to retrieve the URL parameter and map it to the method argument. The @RequestParam annotation is used to bind request parameters to a method parameter in your controller.

@RequestPart

This annotation is used to annotate request handler method arguments. The `@RequestPart` annotation can be used instead of `@RequestParam` to get the content of a specific multipart and bind to the method argument annotated with `@RequestPart`. This annotation takes into consideration the “Content-Type” header in the multipart(request part).

@ResponseBody

This annotation is used to annotate request handler methods. The `@ResponseBody` annotation is similar to the `@RequestBody` annotation. The `@ResponseBody` annotation indicates that the result type should be written straight in the response body in whatever format you specify like JSON or XML. Spring converts the returned object into a response body by using the `HttpMessageConveter`.

@ResponseStatus

This annotation is used on methods and exception classes. `@ResponseStatus` marks a method or exception class with a status code and a reason that must be returned. When the handler method is invoked the status code is set to the HTTP response which overrides the status information provided by any other means. A controller class can also be annotated with `@ResponseStatus` which is then inherited by all `@RequestMapping` methods.

@ControllerAdvice

This annotation is applied at the class level. As explained earlier, for each controller you can use `@ExceptionHandler` on a method that will be called when a given exception occurs. But this handles only those exception that occur within the controller in which it is defined. To overcome this problem you can now use the `@ControllerAdvice` annotation. This annotation is used to define `@ExceptionHandler`, `@InitBinder` and `@ModelAttribute` methods that apply to all `@RequestMapping` methods. Thus if you define the `@ExceptionHandler` annotation on a method in `@ControllerAdvice` class, it will be applied to all the controllers.

@RestController

This annotation is used at the class level. The `@RestController` annotation marks the class as a controller where every method returns a domain object instead of a view. By annotating a class with this annotation you no longer need to add `@ResponseBody` to all the `RequestMapping` method. It means that you no more use view-resolvers or send html in response. You just send the domain object as HTTP response in the format that is understood by the consumers like JSON.

`@RestController` is a convenience annotation which combines `@Controller` and `@ResponseBody`.

@RestControllerAdvice

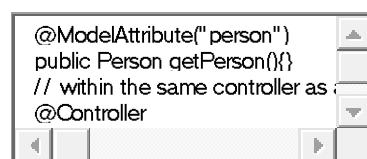
This annotation is applied on Java classes. `@RestControllerAdvice` is a convenience annotation which combines `@ControllerAdvice` and `@ResponseBody`. This annotation is used along with the `@ExceptionHandler` annotation to handle exceptions that occur within the controller.

@SessionAttribute

This annotation is used at method parameter level. The `@SessionAttribute` annotation is used to bind the method parameter to a session attribute. This annotation provides a convenient access to the existing or permanent session attributes.

@SessionAttributes

This annotation is applied at type level for a specific handler. The `@SessionAttributes` annotation is used when you want to add a JavaBean object into a session. This is used when you want to keep the object in session for short lived. `@SessionAttributes` is used in conjunction with `@ModelAttribute`. Consider this example.



```
1 @ModelAttribute("person")
2 public Person getPerson(){ }
3 // within the same controller as above snippet
4 @Controller
5 @SeesionAttributes(value="person", types={Person.class})
6 public class PersonController{ }
```


The @ModelAttribute name is assigned to the @SessionAttributes as value. The @SessionAttributes has two elements. The value element is the name of the session in the model and the types element is the type of session attributes in the model.

Spring Cloud Annotations

@EnableConfigServer

This annotation is used at the class level. When developing a project with a number of services, you need to have a centralized and straightforward manner to configure and retrieve the configurations about all the services that you are going to develop. One advantage of using a centralized config server is that you don't need to carry the burden of remembering where each configuration is distributed across multiple and distributed components.

You can use Spring cloud's @EnableConfigServer annotation to start a config server that the other applications can talk to.

@EnableEurekaServer

This annotation is applied to Java classes. One problem that you may encounter while decomposing your application into microservices is that, it becomes difficult for every service to know the address of every other service it depends on. There comes the discovery service which is responsible for tracking the locations of all other microservices.

Netflix's Eureka is an implementation of a discovery server and integration is provided by Spring Boot. Spring Boot has made it easy to design a Eureka Server by just annotating the entry class with @EnableEurekaServer.

@EnableDiscoveryClient

This annotation is applied to Java classes. In order to tell any application to register itself with Eureka you just need to add the @EnableDiscoveryClient annotation to the application entry point. The application that's now registered with Eureka uses the Spring Cloud Discovery Client abstraction to interrogate the registry for its own host and port.

@EnableCircuitBreaker

This annotation is applied on Java classes that can act as the circuit breaker. The circuit breaker pattern can allow a micro service continue working when a related service fails, preventing the failure from cascading. This also gives the failed service a time to recover.

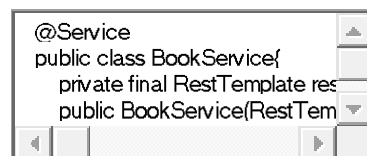
The class annotated with @EnableCircuitBreaker will monitor, open, and close the circuit breaker.

@HystrixCommand

This annotation is used at the method level. Netflix's Hystrix library provides the implementation of Circuit Breaker pattern. When you apply the circuit breaker to a method, Hystrix watches for the failures of the method. Once failures build up to a threshold, Hystrix opens the circuit so that the subsequent calls also fail. Now Hystrix redirects calls to the method and they are passed to the specified fallback methods.

Hystrix looks for any method annotated with the @HystrixCommand annotation and wraps it into a proxy connected to a circuit breaker so that Hystrix can monitor it.

Consider the following example:



```
1  @Service
2  public class BookService{
3      private final RestTemplate restTemplate;
4      public BookService(RestTemplate rest){
5          this.restTemplate = rest;
6      }
7      @HystrixCommand(fallbackMethod = "newList")
8      URI uri = URI.create("http://localhost:8081/recommended");
9      this.restTemplate.getForObject(uri, String.class);
10 }
11 public String newList(){
12     return "Cloud native Java";
13 }
```

```
public String bookList(){
    return
```

Here `@HystrixCommand` is applied to the original method `bookList()`. The `@HystrixCommand` annotation has `newList` as the fallback method. So for some reason if Hystrix opens the circuit on `bookList()`, you will have a placeholder book list ready for the users.



Learn Spring Framework 5 with my [Spring Framework 5: Beginner to Guru](#) course!

Spring Framework Data Access Annotations

`@Transactional`

This annotation is placed before an interface definition, a method on an interface, a class definition, or a public method on a class. The mere presence of `@Transactional` is not enough to activate the transactional behaviour. The `@Transactional` is simply a metadata that can be consumed by some runtime infrastructure. This infrastructure uses the metadata to configure the appropriate beans with transactional behaviour.

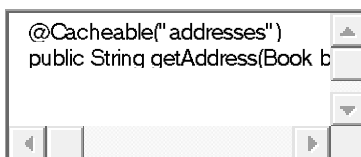
The annotation further supports configuration like:

- The Propagation type of the transaction
- The Isolation level of the transaction
- A timeout for the operation wrapped by the transaction
- A read only flag - a hint for the persistence provider that the transaction must be read only
- The rollback rules for the transaction

Cache-Based Annotations

`@Cacheable`

This annotation is used on methods. The simplest way of enabling the cache behaviour for a method is to annotate it with `@Cacheable` and parameterize it with the name of the cache where the results would be stored.

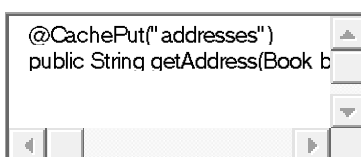


```
1 @Cacheable("addresses")
2 public String getAddress(Book book){ ... }
```

In the snippet above, the method `getAddress` is associated with the cache named `addresses`. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated.

`@CachePut`

This annotation is used on methods. Whenever you need to update the cache without interfering the method execution, you can use the `@CachePut` annotation. That is, the method will always be executed and the result cached.

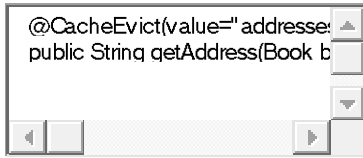


```
1 @CachePut("addresses")
2 public String getAddress(Book book){ ... }
```

Using `@CachePut` and `@Cacheable` on the same method is strongly discouraged as the former forces the execution in order to execute a cache update, the latter causes the method execution to be skipped by using the cache.

@CacheEvict

This annotation is used on methods. It is not that you always want to populate the cache with more and more data. Sometimes you may want remove some cache data so that you can populate the cache with some fresh values. In such a case use the `@CacheEvict` annotation.



```
1 @CacheEvict(value="addresses", allEntries="true")
2 public String getAddress(Book book){ ... }
```

Here an additional element `allEntries` is used along with the cache name to be emptied. It is set to `true` so that it clears all values and prepares to hold new data.

@CacheConfig

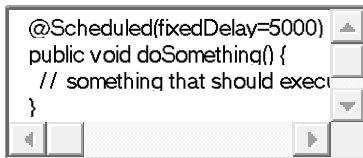
This annotation is a class level annotation. The `@CacheConfig` annotation helps to streamline some of the cache information at one place. Placing this annotation on a class does not turn on any caching operation. This allows you to store the cache configuration at the class level so that you don't have to declare things multiple times.

Task Execution and Scheduling Annotations

@Scheduled

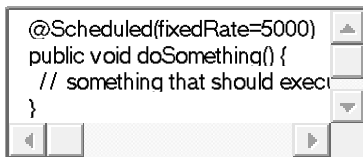
This annotation is a method level annotation. The `@Scheduled` annotation is used on methods along with the trigger metadata. A method with `@Scheduled` should have `void` return type and should not accept any parameters.

There are different ways of using the `@Scheduled` annotation:



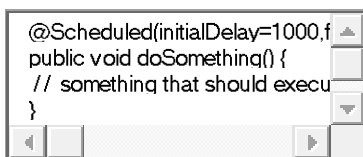
```
1 @Scheduled(fixedDelay=5000)
2 public void doSomething() {
3     // something that should execute periodically
4 }
```

In this case, the duration between the end of last execution and the start of next execution is fixed. The tasks always wait until the previous one is finished.



```
1 @Scheduled(fixedRate=5000)
2 public void doSomething() {
3     // something that should execute periodically
4 }
```

In this case, the beginning of the task execution does not wait for the completion of the previous execution.



```

1 @Scheduled(initialDelay=1000,fixedRate=5000)
2 public void doSomething() {
3 // something that should execute periodically after an initial delay
4 }

```

The task gets executed initially with a delay and then continues with the specified fixed rate.

@Async

This annotation is used on methods to execute each method in a separate thread. The @Async annotation is provided on a method so that the invocation of that method will occur asynchronously. Unlike methods annotated with @Scheduled, the methods annotated with @Async can take arguments. They will be invoked in the normal way by callers at runtime rather than by a scheduled task.

@Async can be used with both void return type methods and the methods that return a value. However methods with return value must have a Future typed return values.

Spring Framework Testing Annotations

@BootstrapWith

This annotation is a class level annotation. The @BootstrapWith annotation is used to configure how the Spring TestContext Framework is bootstrapped. This annotation is used as a metadata to create custom composed annotations and reduce the configuration duplication in a test suite.

@ContextConfiguration

This annotation is a class level annotation that defines a metadata used to determine which configuration files to use to load the ApplicationContext for your test. More specifically @ContextConfiguration declares the annotated classes that will be used to load the context. You can also tell Spring where to locate for the file.

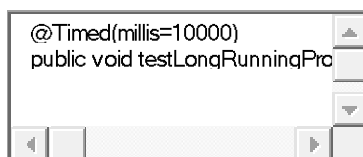
```
@ContextConfiguration(locations={"example/test-context.xml", loader = Custom ContextLoader.class})
```

@WebAppConfiguration

This annotation is a class level annotation. The @WebAppConfiguration is used to declare that the ApplicationContext loaded for an integration test should be a WebApplicationContext. This annotation is used to create the web version of the application context. It is important to note that this annotation must be used with the @ContextConfiguration annotation. The default path to the root of the web application is src/main/webapp. You can override it by passing a different path to the @WebAppConfiguration.

@Timed

This annotation is used on methods. The @Timed annotation indicates that the annotated test method must finish its execution at the specified time period(in milliseconds). If the execution exceeds the specified time in the annotation, the test fails.



```

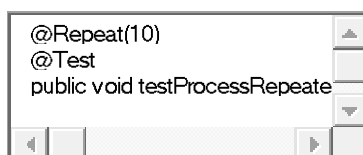
1 @Timed(millis=10000)
2 public void testLongRunningProcess() { ... }

```

In this example, the test will fail if it exceeds 10 seconds of execution.

@Repeat

This annotation is used on test methods. If you want to run a test method several times in a row automatically, you can use the @Repeat annotation. The number of times that test method is to be executed is specified in the annotation.



```
1 @Repeat(10)
2 @Test
3 public void testProcessRepeatedly() { ... }
```

In this example, the test will be executed 10 times.

@Commit

This annotation can be used as both class-level or method-level annotation. After execution of a test method, the transaction of the transactional test method can be committed using the @Commit annotation. This annotation explicitly conveys the intent of the code. When used at the class level, this annotation defines the commit for all test methods within the class. When declared as a method level annotation @Commit specifies the commit for specific test methods overriding the class level commit.

@RollBack

This annotation can be used as both class-level and method-level annotation. The @RollBack annotation indicates whether the transaction of a transactional test method must be rolled back after the test completes its execution. If this true @Rollback(true), the transaction is rolled back. Otherwise, the transaction is committed. @Commit is used instead of @RollBack(false).

When used at the class level, this annotation defines the rollback for all test methods within the class.

When declared as a method level annotation @RollBack specifies the rollback for specific test methods overriding the class level rollback semantics.

@DirtiesContext

This annotation is used as both class-level and method-level annotation. @DirtiesContext indicates that the Spring ApplicationContext has been modified or corrupted in some manner and it should be closed. This will trigger the context reloading before execution of next test. The ApplicationContext is marked as dirty before or after any such annotated method as well as before or after current test class.

The @DirtiesContext annotation supports BEFORE_METHOD, BEFORE_CLASS, and BEFORE_EACH_TEST_METHOD modes for closing the ApplicationContext before a test.

NOTE: Avoid overusing this annotation. It is an expensive operation and if abused, it can really slow down your test suite.

@BeforeTransaction

This annotation is used to annotate void methods in the test class. @BeforeTransaction annotated methods indicate that they should be executed before any transaction starts executing. That means the method annotated with @BeforeTransaction must be executed before any method annotated with @Transactional.

@AfterTransaction

This annotation is used to annotate void methods in the test class. @AfterTransaction annotated methods indicate that they should be executed after a transaction ends for test methods. That means the method annotated with @AfterTransaction must be executed after the method annotated with @Transactional.

@Sql

This annotation can be declared on a test class or test method to run SQL scripts against a database. The @Sql annotation configures the resource path to SQL scripts that should be executed against a given database either before or after an integration test method. When @Sql is used at the method level it will override any @Sql defined in at class level.

@SqlConfig

This annotation is used along with the @Sql annotation. The @SqlConfig annotation defines the metadata that is used to determine how to parse and execute SQL scripts configured via the @Sql annotation. When used at the class-level, this annotation serves as global configuration for all SQL scripts within the test class. But when used directly with the config attribute of @Sql, @SqlConfig serves as a local configuration for SQL scripts declared.

@SqlGroup

This annotation is used on methods. The @SqlGroup annotation is a container annotation that can hold several @Sql annotations. This annotation can declare nested @Sql annotations.

In addition, @SqlGroup is used as a meta-annotation to create custom composed annotations. This annotation can also be used along with repeatable annotations, where @Sql can be declared several times on the same method or class.

@SpringBootTest

This annotation is used to start the Spring context for integration tests. This will bring up the full autoconfiguration context.

@DataJpaTest

The @DataJpaTest annotation will only provide the autoconfiguration required to test Spring Data JPA using an in-memory database such as H2.

This annotation is used instead of @SpringBootTest

@DataMongoTest

The @DataMongoTest will provide a minimal autoconfiguration and an embedded MongoDB for running integration tests with Spring Data MongoDB.

@WebMvcTest

The @WebMvcTest will bring up a mock servlet context for testing the MVC layer. Services and components are not loaded into the context. To provide these dependencies for testing, the @MockBean annotation is typically used.

@AutoConfigureMockMvc

The @AutoConfigureMockMvc annotation works very similar to the @WebMvcTest annotation, but the full Spring Boot context is started.

@MockBean

Creates and injects a Mockito Mock for the given dependency.

@JsonTest

Will limit the auto configuration of Spring Boot to components relevant to processing JSON.

This annotation will also autoconfigure an instance of [JacksonTester](#) or [GsonTester](#).

@TestPropertySource

Class level annotation used to specify property sources for the test class.

Spring Component Scan

[StandardNovember 30, 2017](#)by [jt3 Comments](#)

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

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

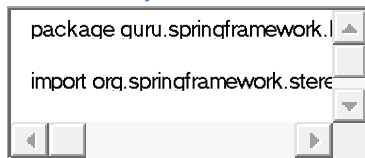
In this post, I will discuss how Spring component scanning works.

Sample Application

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

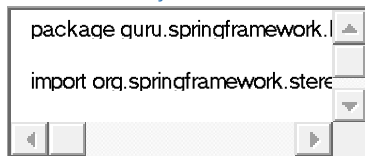
We will start by writing few components.

DemoBeanA.java



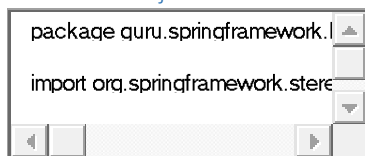
```
1 package guru.springframework.blog.componentscan.example.demopackageA;  
2  
3 import org.springframework.stereotype.Component;  
4  
5 @Component("demoBeanA")  
6 public class DemoBeanA {  
7 }
```

DemoBeanB1.java



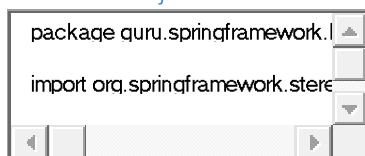
```
1 package guru.springframework.blog.componentscan.example.demopackageB;  
2  
3 import org.springframework.stereotype.Component;  
4  
5 @Component("demoBeanB1")  
6 public class DemoBeanB1 {  
7 }
```

DemoBeanB2.java



```
1 package guru.springframework.blog.componentscan.example.demopackageB;  
2  
3 import org.springframework.stereotype.Component;  
4  
5  
6 @Component("demoBeanB2")  
7 public class DemoBeanB2 extends DemoBeanB1 {  
8 }
```

DemoBeanB3.java

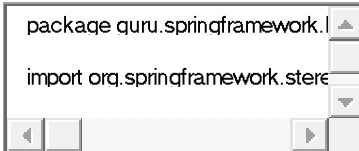


```

1 package guru.springframework.blog.componentscan.example.demopackageB;
2
3 import org.springframework.stereotype.Component;
4
5 @Component("demoBeanB3")
6 public class DemoBeanB3 extends DemoBeanB2{
7 }

```

DemoBeanC.java



```

package guru.springframework.blog.componentscan.example.demopackageC;
import org.springframework.stereotype.Component;

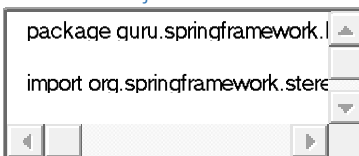
```

```

1 package guru.springframework.blog.componentscan.example.demopackageC;
2
3 import org.springframework.stereotype.Component;
4
5 @Component("demoBeanC")
6 public class DemoBeanC {
7 }

```

DemoBeanD.java



```

package guru.springframework.blog.componentscan.example.demopackageD;
import org.springframework.stereotype.Component;

```

```

1 package guru.springframework.blog.componentscan.example.demopackageD;
2
3 import org.springframework.stereotype.Component;
4
5 @Component("demoBeanD")
6 public class DemoBeanD {
7 }

```

[Click Here to Become a Spring Framework Guru!](#)

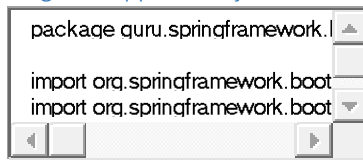
The @SpringBootApplication Annotation

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

To know more about these annotations go through my [Spring Framework Annotations](#) post.

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

The code to create the main class and access components is this.



```

1 package guru.springframework.blog;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.ApplicationContext;
6
7
8 @SpringBootApplication
9 public class BlogPostsApplication {
10
11     public static void main(String[] args) {
12         ApplicationContext context = SpringApplication.run(BlogPostsApplication.class,args);
13         System.out.println("Contains A " +context.
14             containsBeanDefinition("demoBeanA"));
15         System.out.println("Contains B2 " + context.
16             containsBeanDefinition("demoBeanB2"));
17         System.out.println("Contains C " + context.
18             containsBeanDefinition("demoBeanC"));
19
20
21     }
22 }

```

The output of running the main class is this.



As you can notice, all the classes in the sub packages of the main class BlogPostsApplication are auto scanned by Spring.

@ComponentScan – Identifying Base Packages

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

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

BlogPostsApplicationWithComponentScan.java

```

1 package guru.springframework.blog;
2 import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB1;
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.annotation.ComponentScan;
6 import org.springframework.context.annotation.Configuration;
7
8 @Configuration
9 @ComponentScan(basePackages = {"guru.springframework.blog.componentscan.example.demopackageA",
10     "guru.springframework.blog.componentscan.example.demopackageD",
11     "guru.springframework.blog.componentscan.example.demopackageE"},
12     basePackageClasses = DemoBeanB1.class)
13 public class BlogPostsApplicationWithComponentScan {
14     public static void main(String[] args) {
15         ApplicationContext context = SpringApplication.
16             run(BlogPostsApplicationWithComponentScan.class,args);
17         System.out.println("Contains A " +context.
18             containsBeanDefinition("demoBeanA"));
19         System.out.println("Contains B2 " + context.
20             containsBeanDefinition("demoBeanB2"));
21         System.out.println("Contains C " + context.
22             containsBeanDefinition("demoBeanC"));
23         System.out.println("Contains D " + context.
24             containsBeanDefinition("demoBeanD"));
25
26     }
27 }

```

```

Run BlogPostsApplicationWithComponentScan
2017-11-05 21:54:47.434 DEBUG 668 --- [main] o.s.c.e.PropertySourcesPropertyResolver : Could not find key 'spring.liveBeansView.mbeanDomain' in
2017-11-05 21:54:47.438 INFO 668 --- [main] .b.BlogPostsApplicationWithComponentScan : Started BlogPostsApplicationWithComponentScan in 2.535
Contains A true
Contains B2 true
Contains C false
Contains D true
2017-11-05 21:54:47.439 INFO 668 --- [Thread-1] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext:org.springframework.context.annotation.AnnotationConfigApplicationContext@75000000: startup date [2017-11-05 21:54:47.439] root=org.springframework.context.annotation.AnnotationConfigApplicationContext@75000000
2017-11-05 21:54:47.440 DEBUG 668 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Returning cached instance of singleton bean 'lifecycleProcessor'
2017-11-05 21:54:47.441 DEBUG 668 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Destroying singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@5d0a1946: destroying beans [lifecycleProcessor]
Compilation completed successfully in 4s 5ms (a minute ago)
12:57 LF: UTF-8 Git:

```

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

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

Component Scanning Filters

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

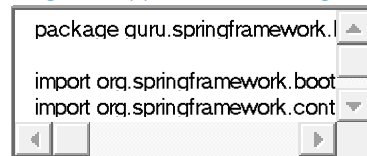
By using filters, you can further narrow the set of candidate components from everything in basePackages to everything in the base packages that matches the given filter or filters.

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

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

The code to disable the default filter is this.

[BlogPostsApplicationDisablingDefaultFilters.java](#)



```
1 package guru.springframework.blog;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.context.annotation.ComponentScan;
5 import org.springframework.context.annotation.Configuration;
6
7
8 @Configuration
9 @ComponentScan(value = "guru.springframework.blog.componentscan.example.demopackageA",
10     useDefaultFilters = false)
11 public class BlogPostsApplicationDisablingDefaultFilters {
12     public static void main(String[] args) {
13         ApplicationContext context = SpringApplication.run(
14             BlogPostsApplicationDisablingDefaultFilters.class, args);
15         System.out.println("Contains A " + context.containsBean("demoBeanA"));
16     }
17 }
```

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

The output on running the main class is this.



```
Run BlogPostsApplicationDisablingDefaultFilters
2017-11-05 22:00:15.878 DEBUG 683 --- [main] o.s.c.e.PropertySourcesPropertyResolver : Searching for key 'spring.liveBeansView.mbeanDomain' in ...
2017-11-05 22:00:15.878 DEBUG 683 --- [main] o.s.c.e.PropertySourcesPropertyResolver : Could not find key 'spring.liveBeansView.mbeanDomain' in ...
2017-11-05 22:00:15.884 INFO 683 --- [main] gPostsApplicationDisablingDefaultFilters : Started BlogPostsApplicationDisablingDefaultFilters in ...
Contains A false
2017-11-05 22:00:15.886 INFO 683 --- [Thread-1] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext: ...
2017-11-05 22:00:15.887 DEBUG 683 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Returning cached instance of singleton bean 'lifecycleP...'
2017-11-05 22:00:15.887 DEBUG 683 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Destroying singletons in org.springframework.beans.factory...
Process finished with exit code 0
Compilation completed successfully in 2s 513ms (2 minutes ago) 466 chars, 10 line breaks 18:1 UTF-8 Git: ma
```

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

Component Scanning Filter Types

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

The available FilterType values are:

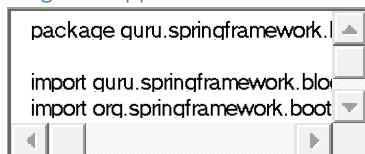
- FilterType.ANNOTATION: Include or exclude those classes with a stereotype annotation
- FilterType.ASPECTJ: Include or exclude classes using an AspectJ type pattern expression
- FilterType.ASSIGNABLE_TYPE: Include or exclude classes that extend or implement this class or interface
- FilterType.REGEX: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

Include Filters

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

The code to use the includeFilters element of @ComponentScan is this.

[BlogPostsApplicationIncludeFilter.java](#)



```
package guru.springframework.blog;
import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;
import org.springframework.boot.SpringApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
```

```
1 package guru.springframework.blog;
2
3 import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.annotation.ComponentScan;
7 import org.springframework.context.annotation.Configuration;
8 import org.springframework.context.annotation.FilterType;
9
10 @Configuration
11 @ComponentScan(basePackages = {"guru.springframework.blog.componentscan.example.demopackageA",
12     "guru.springframework.blog.componentscan.example.demopackageB"},
13     includeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = DemoBeanB2.class),
14     useDefaultFilters = false)
```

```

15 public class BlogPostsApplicationIncludeFilter {
16     public static void main(String[] args) {
17         ApplicationContext context = SpringApplication.
18             run(BlogPostsApplicationIncludeFilter.class,args);
19         System.out.println("Contains A " + context.containsBean("demoBeanA"));
20         System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));
21         System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));
22         System.out.println("Contains B3 " + context.containsBean("demoBeanB3"));
23     }
24 }

```

The output on running the main class is this.

```

Run BlogPostsApplicationIncludeFilter (v1.5.8.RELEASE)
2017-11-22 21:05:32.372 INFO 10628 --- [main] g.s.b.BlogPostsApplicationIncludeFilter : Starting BlogPostsApplicationIncludeFilter on BLR-KRM-STRLT14 with
2017-11-22 21:05:32.378 INFO 10628 --- [main] g.s.b.BlogPostsApplicationIncludeFilter : No active profile set, falling back to default profiles: default
2017-11-22 21:05:32.434 INFO 10628 --- [main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfig
2017-11-22 21:05:32.703 INFO 10628 --- [main] g.s.b.BlogPostsApplicationIncludeFilter : Started BlogPostsApplicationIncludeFilter in 0.809 seconds (JVM ru
Contains A false
Contains B1 false
Contains B2 true
Contains B3 true
2017-11-22 21:05:32.704 INFO 10628 --- [Thread-1] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApp
Process finished with exit code 0
All files are up-to-date (3 minutes ago) 19:107 CRLF: UTF-8: Git: spring-component

```

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

Learn Spring Framework 5 with my Spring Framework 5: Beginner to Guru course!

Include Filters using Regex

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

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

[BlogPostsApplicationFilterTypeRegex.java](#)

```

package guru.springframework.blog;

import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

```

```

1 package guru.springframework.blog;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5 import org.springframework.context.annotation.ComponentScan;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.context.annotation.FilterType;
8
9 @Configuration
10 @ComponentScan(useDefaultFilters = false, includeFilters = @ComponentScan.Filter

```

```

11     (type = FilterType.REGEX, pattern = ".*[A2]"))
12 public class BlogPostsApplicationFilterTypeRegex {
13     public static void main(String[] args) {
14         ApplicationContext context = SpringApplication.
15             run(BlogPostsApplicationFilterTypeRegex.class,args);
16         System.out.println("Contains A " + context.containsBean("demoBeanA"));
17         System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));
18         System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));
19     }
20 }

```

The output of the following code snippet is this.



```

Run BlogPostsApplicationFilterTypeRegex
2017-11-18 18:16:28.911 DEBUG 3084 --- [main] o.s.c.e.PropertySourcesPropertyResolver : Searching for key 'spring.liveBeansView.mbeanDomain' in [applicatio
2017-11-18 18:16:28.911 DEBUG 3084 --- [main] o.s.c.e.PropertySourcesPropertyResolver : Could not find key 'spring.liveBeansView.mbeanDomain' in any proper
2017-11-18 18:16:28.927 INFO 3084 --- [main] .s.b.BlogPostsApplicationFilterTypeRegex : Started BlogPostsApplicationFilterTypeRegex in 0.812 seconds (JVM m
Contains A true
Contains B1 false
Contains B2 true
2017-11-18 18:16:28.927 INFO 3084 --- [Thread-1] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApp
2017-11-18 18:16:28.927 DEBUG 3084 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Returning cached instance of singleton bean 'lifecycleProcessor'
2017-11-18 18:16:28.927 DEBUG 3084 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Destroying singletons in org.springframework.beans.factory.support.
Compilation completed successfully in 1s 297ms (a minute ago)

```

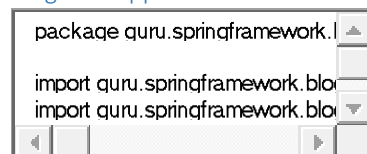
As shown in the preceding figure, the classes whose names end with A, or 2 are detected by Spring.

Exclude Filters

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

The code to use an exclude filter is this.

[BlogPostsApplicationExcludeFilter.java](#)



```

package guru.springframework.blog;

import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB1;
import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;

```

```

1 package guru.springframework.blog;
2
3 import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB1;
4 import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.context.ApplicationContext;
7 import org.springframework.context.annotation.ComponentScan;
8 import org.springframework.context.annotation.Configuration;
9 import org.springframework.context.annotation.FilterType;
10
11 @Configuration
12 @ComponentScan(basePackageClasses = {DemoBeanB1.class},
13     excludeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE,

```

```

14         value = DemoBeanB2.class))
15 public class BlogPostsApplicationExcludeFilter {
16     public static void main(String[] args) {
17         ApplicationContext context = SpringApplication.
18             run(BlogPostsApplicationExcludeFilter.class,args);
19         System.out.println("Contains B1 " + context.containsBean("demoBeanB1"));
20         System.out.println("Contains B2 " + context.containsBean("demoBeanB2"));
21     }
22 }

```

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

The output is this.



```

Run BlogPostsApplicationExcludeFilter
2017-11-05 22:04:03.880 DEBUG 701 --- [main] o.s.c.e.PropertySourcesPropertyResolver : Could not find key 'spring.livebeansview.mbeanDomain' in
2017-11-05 22:04:03.883 INFO 701 --- [main] g.s.b.BlogPostsApplicationExcludeFilter : Started BlogPostsApplicationExcludeFilter in 2.104 seconds
Contains B1 true
Contains B2 false
2017-11-05 22:04:03.884 INFO 701 --- [Thread-1] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext: org.springframework.context.annotation.AnnotationConfigApplicationContext@701
2017-11-05 22:04:03.885 DEBUG 701 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Returning cached instance of singleton bean 'lifecycleProcessor'
2017-11-05 22:04:03.885 DEBUG 701 --- [Thread-1] o.s.b.f.s.DefaultListableBeanFactory : Destroying singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@701: destroying beans with names [org.springframework.context.annotation.AnnotationConfigApplicationContext@701]
Process finished with exit code 0
Compilation completed successfully in 2s 546ms (a minute ago) 22:2 LF: UTF-8 Git: ma

```

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

Summary

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

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