Name: ANURAG KUMAR

Emp id: INN912

**SPRING ANNOTATIONS**

Spring Framework provides a rich set of annotations to facilitate various aspects of application development. Annotations help by providing metadata and instructions to the Spring container. They allow the container to understand and configure various components within a Spring application. To make it more precise, annotations help during component scanning, dependency injection, auto wiring, defining configurations, aspect-oriented programming, data access, testing and many more areas.  
  
**BEAN ANNOTATIONS**

A **bean** is an object that is instantiated, assembled, and otherwise managed by the Spring IoC (Inversion of Control) container. Beans are the backbone of any Spring-based application, and they are defined in the Spring container in several ways, such as via XML configuration, annotations, or Java-based configuration.  
  
**1. @Bean**

The *@Bean* is a method-level annotation used to declare a spring bean. When the container executes the annotated method, it registers the return value as a bean within a *BeanFactory*.

By default, the bean name will be the same as the method name. To customize the bean name, we use its ‘*name*’  or  ‘*value*‘ attribute.

@Bean

EmployeeService employeeService()

{

return new EmployeeServiceImpl();

}

**2. @Component, @Controller, @Repository, @Service**

These annotations are called stereotype annotations. When component scanning is enabled, Spring will automatically import these beans into the container and inject them into dependencies.

* The @Component annotation is a generic annotation and marks a Java class as a bean.
* The @Controller annotation marks a class as a Spring MVC controller.
* The @Repository annotation is a specialization of the @Component annotation. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions (thrown from DAO methods) eligible for translation into Spring DataAccessException.
* The @Service annotation is also a specialization of @Component and used over service-layer classes because it specifies intent better. It doesn’t currently provide any additional behavior.

@Controller

public class UserMgmtController {

*//API handler methods*

}

@Service

public class EmployeeServiceImpl inplements EmployeeService {

*//...*

}

@Repository

public class EmployeeDAOImpl implements EmployeeDAO {

...

}

**3. @Qualifier**

During autowiring, if more than one bean of the same type is available in the container then the container will throw runtime exception. To fix this problem, we have to specifically tell Spring which bean has to be injected using this annotation.

public class EmployeeService {

@Autowired

@Qualifier("fsRepository")

private Repository repository;

}

Here, the bean with the name ‘fsRepository’ will be injected.

**4. @Autowired**

This annotation marks a constructor, field, setter method, or config method as to be autowired by dependency injection. We can mark whether the annotated dependency is required (mandatory to populate) or not using it’s ‘required’ attribute. By default, its value is ‘true’.

@Service

public class EmployeeService {

@Autowired

private EmployeeDao dao;

}

**5. @Value**

Applicable at the field or method/constructor parameter level, and indicates a default value expression for the affected argument.

public class SomeService {

@Value("${ENV:DEV}")

private String environment;

}

**6. @Lazy**

Indicates whether a bean is to be lazily initialized. By default, in spring DI, eager initialization will occur. When applied over any bean, initialization of that bean will not happen until referenced by another bean or explicitly retrieved from the enclosing *BeanFactory*.

@Component

public class SomeService {

@Autowired

@Lazy

private RemoteService remoting;

}

**7. @DependsOn**

During component scanning, it is used to specify the beans on which the current bean depends on. The specified beans are guaranteed to be created by the container before this bean.

@Component

public class SomeService {

@Autowired

@DependsOn ("pingService")

private RemoteService remoting;

}

**8. @LookUp**

Indicates a method as a lookup method. It is best used for injecting a prototype-scoped bean into a singleton bean.

@Component

@Scope("prototype")

public class AppNotification {

*//prototype-scoped bean*

}

@Component

public class NotificationService {

@Lookup

public AppNotification getNotification() {

*//return new AppNotification();*

}

}

**9. @Primary**

Indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency. When not using *@Primary*, we may need to provide *@Qualifier* annotation to correctly inject the beans.

In the given example, when *FooRepository* will be autowired, the instance of *HibernateFooRepository* will be injected – until *@Qualifier* annotation is used.

@Component

public class JdbcFooRepository

extends FooRepository {

}

@Primary

@Component

public class HibernateFooRepository

extends FooRepository {

}

**10. @Scope**

Indicates the name of a scope to use for instances of the annotated type. In Spring, beans can be in one of six bean scopes i.e. singleton, prototype, request, session, application and WebSocket. Note that the default scope is ‘*singleton*‘.

@Bean

@Scope("prototype")

public class BeanClass {

return new BeanClass();

}

**CONFIGURATION ANNOTATIONS**

They control how the other beans are discovered and the scope where they can be used.

1. **@ComponentScan**

@ComponentScan along with @Configuration is used to enable and configure component scanning. By default, if we do not specify the path, it scans the current package and all of its sub-packages for components.

Using component scanning, spring can auto-scan all classes annotated with the stereotype annotations @Component, @Controller, @Service and @Repository and configure them with BeanFactory.

@Configuration

@ComponentScan(basePackages = {com.howtodoinjava.data.jpa})

public class JpaConfig {

}

1. **@Configuration**

It indicates that a class declares one or more *@Bean* methods and can be processed by the container to generate bean definitions when used along with *@ComponentScan*.

@Configuration

public class AppConfig {

@Bean

public AppUtils appUtils()

{

return new AppUnits();

}

}

1. **@Profile**

It indicates that a component is eligible for bean registration when one or more specified profiles are active. A profile is a named logical grouping of beans e.g. dev, prod etc.

@Profile("dev")

@Configuration

public class DevelopmentProfileConfiguration {

@Bean

public DataSource mySqlDataSource() {*/\*...\*/*}

}

1. **@Import and @ImportResource**

The *@Import* annotation indicates one or more component classes to import – typically *@Configuration* classes. The *@Bean* definitions declared in imported *@Configuration* classes should be accessed by using *@Autowired* injection.

@Configuration

@Import({ JpaConfig.class, SchedulerConfig.class })

public class AppConfig {

}

**SPRING BOOT ANNOTATIONS**

**@EnableAutoConfiguration**

 It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. @SpringBootApplication.

**@SpringBootApplication**

It is a combination of three annotations @EnableAutoConfiguration, @ComponentScan, and @Configuration.

**SPRING MVC and REST ANNOTATIONS**

**@RequestMapping**

It is used to map the web requests. It has many optional elements like consumes, header, method, name, params, path, produces, and value. We use it with the class as well as the method.

@Controller

public class BooksController

{

@RequestMapping("/computer-science/books")

public String getAllBooks(Model model)

{

//application code

return "bookList";

}

**@GetMapping**

 It maps the HTTP GET requests on the specific handler method. Itis used to create a web service endpoint that fetches It is used instead of using: @RequestMapping(method = RequestMethod.GET)

**@PostMapping**

 It maps the HTTP POST requests on the specific handler method. It is used to create a web service endpoint that creates It is used instead of using: @RequestMapping(method = RequestMethod.POST)

**@PutMapping**

It maps the HTTP PUT requests on the specific handler method. It is used to create a web service endpoint that creates or updates It is used instead of using: @RequestMapping(method = RequestMethod.PUT)

**@DeleteMapping**

 It maps the HTTP DELETE requests on the specific handler method. It is used to create a web service endpoint that deletes a resource. It is used instead of using: @RequestMapping(method = RequestMethod.DELETE)

**@PatchMapping**

 It maps the HTTP PATCH requests on the specific handler method. It is used instead of using: @RequestMapping(method = RequestMethod.PATCH)

**@RequestBody**

 It is used to bind HTTP request with an object in a method parameter. Internally it uses HTTP MessageConverters to convert the body of the request. When we annotate a method parameter with @RequestBody, the Spring framework binds the incoming HTTP request body to that parameter.

**@ResponseBody**

It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

**@PathVariable**

It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.

**@RequestParam**

 It is used to extract the query parameters form the URL. It is also known as a query parameter. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.

**@RequestHeader**

It is used to get the details about the HTTP request headers. We use this annotation as a method parameter. The optional elements of the annotation are name, required, value, defaultValue. For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

**@RestController**

 It can be considered as a combination of @Controller and @ResponseBody annotations. The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.

**@RequestAttribute**

It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.