Spring Test Framework

Spring Boot Annotations

Spring Boot made configuring Spring easier with its auto-configuration feature.

@SpringBootApplication

We use this annotation to **mark the main class of a Spring Boot application**:

```
@SpringBootApplication
class VehicleFactoryApplication {
  public static void main(String[] args) {
    SpringApplication.run(VehicleFactoryApplication.class, args);
  }
}
```

@SpringBootApplication encapsulates ***@Configuration***, ***@EnableAutoConfiguration***, **and *@ComponentScan*** annotations with their default attributes.

@**Required**: It applies to the bean setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception BeanInitilizationException.

@Autowired: Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

@Configuration: It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

@ComponentScan: It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

@Bean: It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.

@Component: It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configure it in the application context as a Spring Bean.

@Service: It is also used at class level. It tells the Spring that class contains the business logic.

@Repository: It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

@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 Annotations:

- **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is 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.

@EnableAutoConfiguration

@EnableAutoConfiguration, as its name says, enables auto-configuration. It means that **Spring Boot looks for auto-configuration beans** on its classpath and automatically applies them.

Note, that we have to use this annotation with *@Configuration*:

```
@Configuration
@EnableAutoConfiguration
class VehicleFactoryConfig {}
```

@ConditionalOnClass and *@ConditionalOnMissingClass*

Using these conditions, Spring will only use the marked auto-configuration bean if the class in the annotation's **argument is present/absent**:

```
@Configuration
@ConditionalOnClass(DataSource.class)
class MySQLAutoconfiguration {    //...}
```

@ConditionalOnBean and *@ConditionalOnMissingBean*

We can use these annotations when we want to define conditions based on the **presence or absence of a specific bean**:

```
@Bean
@ConditionalOnBean(name = "dataSource")
LocalContainerEntityManagerFactoryBean entityManagerFactory() {    // ...}
```

@ConditionalOnProperty

With this annotation, we can make conditions on the **values of properties**:

```
@Bean
@ConditionalOnProperty(    name = "usemysql",     havingValue = "local")
DataSource dataSource() {    // ...}
```

@ConditionalOnResource

We can make Spring to use a definition only when a specific **resource is present**:

```
@ConditionalOnResource(resources = "classpath:mysql.properties")
Properties additionalProperties() {    // ...}
```

@ConditionalOnWebApplication and *@ConditionalOnNotWebApplication*

With these annotations, we can create conditions based on if the current **application is or isn't a web application**:

```
@ConditionalOnWebApplication
HealthCheckController healthCheckController() {    // ...}
```

@ConditionalExpression

We can use this annotation in more complex situations. Spring will use the marked definition when the **SpEL expression is evaluated to true**:

```
@Bean
@ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}")
DataSource dataSource() {    // ...}
```

@Conditional

For even more complex conditions, we can create a class evaluating the **custom condition**. We tell Spring to use this custom condition with *@Conditional*:

```
@Conditional(HibernateCondition.class)
Properties additionalProperties() {    //...}
```