Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide **supplemental** information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

# Spring Boot Annotations

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

### @EnableAutoConfiguration:

It auto-configures the bean that is present in the **classpath**, **existing beans**, and **properties** and configures it to run the methods.

#### Internal working:

**1. Annotation Metadata Processing**

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
@AutoConfigurationPackage  
@Import(AutoConfigurationImportSelector.class)  
public @interface EnableAutoConfiguration {  
 ...  
}

* **Key point**: It uses @Import(AutoConfigurationImportSelector.class)

This means Spring will load all beans and configs selected by AutoConfigurationImportSelector.

**2. AutoConfigurationImportSelector**

This is the **core engine** of auto-configuration.

It:

* Reads a file called:

META-INF/spring.factories (Spring Boot 2.x)  
OR  
META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports (Spring Boot 3.x)

It looks for entries under:

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
com.example.XAutoConfiguration,\  
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\  
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\  
 ...

* These classes are then **imported** into the Spring container just like if you had added @Import(...) manually.

**3. Conditional Annotations**

Each auto-config class is loaded **only if certain conditions are met** — thanks to @Conditional... annotations.

@Configuration  
@ConditionalOnClass(DataSource.class)  
@ConditionalOnMissingBean(DataSource.class)  
public class DataSourceAutoConfiguration {  
 @Bean  
 public DataSource dataSource() {  
 // Create and return DataSource bean  
 }  
}

This means:

* Load only if DataSource.class is on the classpath
* And no other DataSource bean already exists

#### 🔍 Summary Flow of @EnableAutoConfiguration

@SpringBootApplication  
 ↓  
@EnableAutoConfiguration  
 ↓  
@Import(AutoConfigurationImportSelector.class)  
 ↓  
Reads spring.factories or AutoConfiguration.imports  
 ↓  
Loads matching auto-config classes conditionally  
 ↓  
Registers beans into ApplicationContext

##### What happens when you exclude specific packages or configurations?

You can **selectively disable** certain auto-configurations using:

**🔹 Option 1: Exclude specific classes**

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

public class MyApp { }

🟢 Result:

* DataSourceAutoConfiguration won't be applied
* Even if you include spring-boot-starter-data-jpa, no DataSource will be configured

**🔹 Option 2: Use @EnableAutoConfiguration(exclude = ...) directly**

@EnableAutoConfiguration(exclude = {WebMvcAutoConfiguration.class})

public class MyApp { }

🟢 Result:

* Spring won’t configure DispatcherServlet, ViewResolvers, or other Web MVC stuff

**🔹 Option 3: Exclude by property**

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

🟢 Result:

* Same effect as using exclude in annotation — but controlled via properties

**🔹 Option 4: Use @ConditionalOnMissingClass or @ConditionalOnProperty inside custom configs to selectively disable auto-config classes**

**🧠 Why would you exclude auto-configurations?**

1. **Custom Configuration**:  
   You want to define your own DataSource, SecurityFilterChain, etc.
2. **Optimize Startup Time**:  
   Exclude features you don’t use (like JMX, Actuator, WebSocket, etc.)
3. **Use Alternative Libraries**:  
   You want to configure your own Kafka client, instead of using Spring Boot’s auto-config

### @Configuration:

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

#### Internal Working :

 **Annotation Processing**:

* Spring’s **ConfigurationClassPostProcessor** detects classes annotated with @Configuration during startup.
* It processes them to identify @Bean methods.

 **Enhancement via CGLIB**:

* Spring **creates a subclass proxy** (using **CGLIB**) of the @Configuration class.
* This ensures that calling one @Bean method from another doesn’t create a new object every time — it retrieves the **singleton** from the container.

 **Method Interception**:

* When a method annotated with @Bean is called inside another @Bean method, Spring **intercepts** the call and returns the existing bean from the container, not a new one.

@Configuration  
public class AppConfig {  
  
 @Bean  
 public Engine engine() {  
 return new Engine();  
 }  
  
 @Bean  
 public Car car() {  
 return new Car(engine()); // NOT a new Engine()  
 }  
}

Internally Spring does:

public class AppConfig$$EnhancerBySpringCGLIB extends AppConfig {  
 private Engine engine;  
  
 public Engine engine() {  
 if (this.engine == null) {  
 this.engine = super.engine();  
 }  
 return this.engine;  
 }  
  
 public Car car() {  
 return new Car(engine()); // returns cached Engine  
 }  
}

Without this enhancement, calling engine() inside car() would create a **new instance every time**, violating Spring’s singleton behavior.

**⚙️ Behind the Scenes: Internal Classes Involved**

| **Component** | **Role** |
| --- | --- |
| @Configuration | Marks config class |
| @Bean | Marks factory methods |
| ConfigurationClassPostProcessor | Detects and processes @Configuration classes |
| CGLIB | Used to proxy/enhance config classes |
| BeanDefinition | Metadata created for each @Bean method |
| ApplicationContext | Stores and manages the beans |

**❗What happens if you remove @Configuration?**

Let’s say you only keep @Component instead of @Configuration:

💥 **Problem:** Each call to engine() creates a new object, because CGLIB enhancement is not applied. So car() gets a **new Engine**, not the singleton one — **unexpected behavior**.

**🧠 In Spring, a Proxy is used for:**

| **Feature** | **Proxy Purpose** |
| --- | --- |
| @Transactional | Opens/closes transactions around method calls |
| @Async | Runs methods in a new thread |
| @Configuration | Ensures singleton @Bean methods return the same instance |
| AOP (Aspect-Oriented) | Used for logging, security, performance timing, etc. |

**❓ Does Spring Create a Proxy for @Scope("prototype") Beans?**

**✅ By default:**

**No**, Spring does **not** create a proxy for prototype beans **unless** you explicitly ask for it.

| **Feature** | **Explanation** |
| --- | --- |
| Default behavior | Prototype beans are not proxied |
| Problem | Injecting prototype bean into singleton results in only one instance |
| Solution | Use @Scope(proxyMode = ...) to enable proxy-based injection |
| Alternative | Use ObjectFactory, Provider, or manual lookup from context |
| Proxy type | CGLIB for classes (TARGET\_CLASS), JDK proxy for interfaces (INTERFACES) |

### @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.

**🔍 Internal Working of @ComponentScan**

**1. Annotation Declaration**

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
@Repeatable(ComponentScans.class)  
public @interface ComponentScan {  
 String[] basePackages() default {};  
 Class<?>[] basePackageClasses() default {};  
 ...  
}

You can control:

* basePackages: specific packages to scan
* basePackageClasses: specific classes to infer packages from
* excludeFilters: classes to exclude
* includeFilters: narrow down scanning targets

**2. When Spring Boot Starts (or AnnotationConfigApplicationContext in plain Spring):**

* @ComponentScan is picked up by a special processor:  
  🔧 ClassPathBeanDefinitionScanner
* It performs a **classpath scan** to find all classes with stereotypes like:
  + @Component
  + @Service
  + @Repository
  + @Controller
  + @Configuration

**3. Each candidate class is read and converted into a:**

* BeanDefinition
* Which is then registered with the **ApplicationContext**

**🧠 What if you don’t use @ComponentScan?**

If you define a Spring Boot app in a subpackage of your beans:

@SpringBootApplication

public class MainApp { } // in com.example.main

But your beans are in com.example.beans (not a subpackage), then those beans **will not be scanned**.

You must explicitly specify:

@ComponentScan(basePackages = "com.example.beans")

# Core Spring Framework Annotations

**@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**.

**Example**

public class Machine  
{  
 private Integer cost;  
 @Required  
 public void setCost(Integer cost)  
 {  
 this.cost = cost;  
 }  
 public Integer getCost()  
 {  
 return cost;  
 }  
}

**@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.

**Example**

@Component  
public class Customer  
{  
 private Person person;  
 @Autowired  
 public Customer(Person person)  
 {  
 this.person=person;  
 }  
}

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

**Example**

@Configuration  
public class Vehicle  
{  
 @BeanVehicle engine()  
 {  
 return new Vehicle();  
 }  
}

**@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.

**Example**

@ComponentScan(basePackages = "com.javatpoint")  
@Configuration  
public class ScanComponent  
{  
// ...   
}

**@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.

**Example**

@Bean  
public BeanExample beanExample()  
{  
 return new BeanExample ();  
}

# Spring Framework Stereotype Annotations

**@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**.

**Example**

@Component  
public class Student  
{   
.......  
}

**@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

**Example**

@Controller  
@RequestMapping("books")  
public class BooksController  
{  
 @RequestMapping(value = "/{name}", method = RequestMethod.GET)  
 public Employee getBooksByName()  
 {  
 return booksTemplate;  
 }  
}

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

**Example**

package com.javatpoint;  
@Service  
public class TestService  
{  
 public void service1()  
 {  
//business code   
 }  
}

**@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.

package com.javatpoint;  
@Repository  
public class TestRepository  
{  
 public void delete()  
 {  
//persistence code   
 }  
}

# 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.

**Example**

@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. 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.

## 🔍 Important Conditional Annotations Used Inside AutoConfigs:

| **Annotation** | **Description** |
| --- | --- |
| @ConditionalOnClass | Only apply config if a class is on the classpath |
| @ConditionalOnMissingBean | Only apply config if a bean is not already defined |
| @ConditionalOnProperty | Enable config if a specific property is set |
| @ConditionalOnWebApplication | Apply config only in a web app |
| @ConditionalOnBean | Apply config only if a specific bean exists |