I'm not a robot

Premium Read: Access my best content on Medium member-only articles — deep dives into Java, Spring Boot, Microservices, backend architecture, interview preparation, career advice, and industry-standard best practices. Some premium posts are free to read — no account needed. Follow me on Medium to stay updated and support my writing. Top 10 Udemy Courses (Huge Discount): Explore My Udemy Courses — Learn through real-time, project-based development. ▶ Subscribe to My YouTube Channel (172K+ subscribers): Java Guides on YouTube In this tutorial, we will learn how to use @Value Annotation in Spring or Spring boot applications.YouTube VideoSpring @Value annotation is used to assign default values to variables and method arguments.The @Value annotation is mostly used to get value for specific property keys from the properties file.We can read spring environment variables as well as system variables using @Value annotation.In this example, we will see how to use @Value annotation to read the value for a property key, and read environment variables as well as system variables.Add the below properties to the application.properties file:mail.email=javaguides@gmail.com mail.host=gmail.com mail.password=passwordNext, we will see how to read the values for the above property keys using @Value annotation.Using @Value Annotation:Create a ValueAnnotationDemo class and add the following to it:import org.springframework.beans.factory.annotation.Value; import org.springframework.stereotype.Component; @Component public class ValueAnnotationDemo { @Value("default value") private String testDefaultValue; @Value("${java.home}") private String javaHome; @Value("${HOME}") private String homeDir; @Value("${mail.email}") private String email; @Value("${mail.host}") private String host; @Value("${mail.password}") private String password; public String getTestDefaultValue() { return testDefaultValue; } public String getJavaHome() { return javaHome; } public String getHomeDir() { return homeDir; } public String getEmail() { return email; } public String getHost() { return host; } public String getPassword() { return password; } }From the above code, Spring @Value annotation is used to assign a default value to the variable: @Value("default value") private String testDefaultValue;Spring @Value annotation used to read the property values: @Value("${mail.email}") private String email; @Value("${mail.host}") private String host; @Value("${mail.password}") private String password;Reading spring environment variables as well as system variables using @Value annotation: @Value("${java.home}") private String javaHome; @Value("${HOME}") private String homeDir;Testing:Next, let's test the usage of @Value annotation:import net.javaguides.springboot.config.ValueAnnotationDemo; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; @SpringBootApplication public class Application { public static void main(String[] args) { var context = SpringApplication.run(SpringbootDockerDemoApplication.class, args); ValueAnnotationDemo valueAnnotationDemo = context.getBean(ValueAnnotationDemo.class); System.out.println(valueAnnotationDemo.getTestDefaultValue()); System.out.println(valueAnnotationDemo.getHost()); System.out.println(valueAnnotationDemo.getEmail()); System.out.println(valueAnnotationDemo.getPassword()); System.out.println(valueAnnotationDemo.getJavaHome()); System.out.println(valueAnnotationDemo.getEmail()); System.out.println(valueAnnotationDemo.getHomeDir()); } }Output:default value gmail.com javaguides@gmail.com password /Users/rameshfadatare/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home javaguides@gmail.com /Users/rameshfadatare Related Spring and Spring Boot Annotations The Hidden Magic of Spring Boot: Secrets Every Developer Should Know What Happens When You Hit a Spring Boot REST API Endpoint (Behind the Scenes) Spring Boot Exception Handling Build CRUD REST API with Spring Boot, Spring Data JPA, Hibernate, and MySQL Spring Boot DELETE REST API: @DeleteMapping Annotation Spring Boot PUT REST API—@PutMapping Annotation Spring Boot POST REST API Spring Boot GET REST API—@GetMapping Annotation Spring Boot REST API with Request Param | Spring Boot Course Spring Boot REST API with Path Variable—@PathVariable Chapter 13: Understanding @SpringBootApplication Annotation | Spring Boot Course Chapter 5: Create Spring Boot Project and Build Hello World REST API | Spring Boot Course 10 Real-World Spring Boot Architecture Tips Every Developer Should Follow Top 10 Spring Boot Tricks Every Java Developer Should Know Debugging Spring Dependency Injection Issues - Very Important Common Code Smells in Spring Applications—How to Fix Them Spring Boot + OpenAI ChatGPT API Integration Tutorial Spring Boot Course -> New Series on Medium ❤ Spring Boot Microservices with RabbitMQ Example React JS + Spring Boot Microservices Dockerizing a Spring Boot Application How to Change the Default Port in Spring Boot How to Change Context Path in Spring Boot Top 10 Spring Boot REST API Mistakes and How to Avoid Them (2025 Update) Spring Boot REST API Best Practices Spring Boot Security Database Authentication Example Tutorial Spring Boot Security Form-Based Authentication Spring Boot Security In-Memory Authentication What is Spring Boot Really All About? Why Spring Boot over Spring? Top 10 Spring Boot Key Features That You Should Know Spring vs Spring Boot Setting Up the Development Environment for Spring Boot Spring Boot Auto-Configuration: A Quick Guide Spring Boot Starters Quick Guide to Spring Boot Parent Starter Spring Boot Embedded Servers Spring Boot Thymeleaf Hello World Example Chapter 10: Spring Boot DevTools | Spring Boot Course Chapter 13: Spring Boot REST API That Returns JSON | Spring Boot Course Spring Boot REST API That Returns List of Java Objects in JSON Format Top 10 Spring Boot Mistakes and How to Avoid Them Advanced Spring Boot Concepts that Every Java Developer Should Know What Are Microservices in Spring Boot? Integrating React Frontend with Spring Boot ChatGPT API (Step-by-Step Guide) Build a Chatbot Using Spring Boot, React JS, and ChatGPT API Top 10 Mistakes in Spring Boot Microservices and How to Avoid Them (With Examples) Spring Boot Security Best Practices: Protecting Your Application from Attacks 🔒 Dependency Injection in Spring (Explained with Coding Examples) ⚙ How Spring Container Works Behind the Scenes How Spring Container Works Behind the Scenes (Spring Container Secrets Revealed!) Spring @Component vs @Bean vs @Service vs @Repository Explained How Component Scanning Works Behind the Scenes in Spring How Spring Autowiring Works Internally Top 20 Spring Boot Best Practices for Java Developers Build Spring Boot React Full Stack Project—Todo App [2025 Update] Spring vs Spring MVC vs Spring Boot Spring Boot Best Practices: Use DTOs Instead of Entities in API Responses Spring Boot DTO Tutorial (Using Java record) – Complete CRUD REST API Implementation Spring Boot Architecture: Controller, Service, Repository, Database and Architecture Flow Java Stream filter() Method with Real-World Examples Spring Boot Auto Configuration Explained | How It Works Spring Boot Profiles: How to Manage Environment-Based Configurations Create a Custom Spring Boot Starter | Step-by-Step Guide Spring Boot Starter Modules Explained | Auto-Configuration Guide Deploy Spring Boot Applications with Profile-Based Settings | Step-by-Step Guide Spring Boot Performance Tuning: 10 Best Practices for High Performance Spring Boot @ComponentScan Annotation | Customizing Component Scanning Difference Between @RestController and @RequestMapping in Spring Boot Spring Boot @Cacheable Annotation – Improve Performance with Caching Spring Boot Redis Cache—@Cacheable Complete Guide When to Use @Service, @Repository, @Controller, and @Component Annotations in Spring Boot Why, When, and How to Use @Bean Annotation in Spring Boot App Java Spring Boot vs. Go (Golang) for Backend Development in 2025 Is Autowired Annotation Deprecated in Spring Boot? Everything You Need to Know Stop Making These Common Mistakes in Spring Boot Projects Top 10 Mind-Blowing Spring Boot Tricks for Beginners Why Choose Spring Boot Over Spring Framework? | Key Differences and Benefits How to Run a Spring Boot Application | 5 Easy Ways for Developers What is AutoConfiguration in Spring Boot? | Explained with Example Customize Default Configuration in Spring Boot | 5 Proven Ways Chapter 12: Understanding SpringApplication.run() Method Internals | Spring Boot Course What is CommandLineRunner in Spring Boot? How to Create Custom Bean Validation in Spring Boot Can You Build a Non-Web Application with Spring Boot? How to Disable Auto-Configuration in Spring Boot (Step-by-Step Guide) Top 25 Spring Boot Interview Questions and Answers for Beginners How to Use Java Records with Spring Boot Spring Boot Constructor Injection Explained with Step-by-Step Example Stop Using @Transactional Everywhere: Understand When You Actually Need It Stop Writing Fat Controllers: Follow the Controller-Service-Repository Pattern Stop Using Field Injection in Spring Boot: Use Constructor Injection Stop Sharing Databases Between Microservices: Use Database Per Service Pattern 10 Java Microservices Best Practices Every Developer Should Follow How to Choose the Right Java Microservices Communication Style (Sync vs Async) How to Implement Event-Driven Communication in Java Microservices (Step-by-Step Guide with Kafka) Stop Building Tight-Coupled Microservices: Aim for Loose Coupling Spring Boot Microservices E-Commerce Project: Step-by-Step Guide Spring Boot Microservices with RabbitMQ Example React JS + Spring Boot Microservices The Ultimate Microservices Roadmap for Beginners: Building Modern Scalable Systems What Are Microservices in Spring Boot? Top 5 Message Brokers Every Developer Should Know Top 10 Spring Cloud Microservices Best Practices [Removed Deprecated Features] Best Tools for Microservices Development in 2025 How to Break a Monolithic Application into Microservices (E-Commerce Use Case) Monoliths Aren't Dead—Microservices Are Just Overused When to Break a Monolith: A Developer's Checklist Java Is Still the King of Microservices—And Here's the Proof 5 Microservices Design Patterns You Must Know in 2025 Bulkhead Pattern in Microservices—Improve Resilience and Fault Isolation Strangler Fig Pattern in Microservices—Migrate Monolith to Microservices Event Sourcing Pattern in Microservices (With Real-World Example) Circuit Breaker Pattern in Microservices using Spring Boot 3, WebClient and Resilience4j CQRS Pattern in Microservices Aggregator Design Pattern in Microservices—A Complete Guide Database Per Service Pattern in Microservices API Gateway Pattern in Microservices—A Complete Guide Saga Pattern in Microservices: A Step-by-Step Guide Microservices Are a Mess Without These Java Design Patterns Java Microservices Interview Questions and Answers for Freshers Top Microservices Interview Questions and Answers for Experienced Professionals Top 10 Microservices Design Pattern Interview Questions and Answers Top Microservices Tricky Interview Questions You Should Know (With Answers) Microservices Best Practices: Building Scalable and Resilient Systems Why Microservices Are the Future of Software Architecture Microservices with Spring Cloud: Simplify Your Architecture Spring Boot and Microservices Roadmap for Beginners [2025 Update] Best Programming Language for Microservices Project Development in 2025 My 50+ Must-Read Microservices Tutorials, Articles and Guides on the Medium Platform The main focus of this article is to help you understand how Spring's @Value annotation works. @Value is a Java annotation that is used at the field or method/constructor parameter level and it indicates a default value for the affected argument. It is commonly used for injecting values into configuration variables - which we will show and explain in the next part of the article. Basic Assignment For the easiest examples, we will assign values to three different fields using the @Value annotation by giving them explicit values: private String trainee; private int hoursOfCode; private boolean passedAssesmentTest; It's very important to note that the argument passed to the @Value annotation can only be a String. Spring will convert the value to the specified type and the assignment will be done without any problems - even if we are passing String values to int or boolean variables. Spring Environment Injecting values from properties files with the help of @Value annotation is probably the most used use-case in real-life applications. We will use the default property file for Spring Boot - application.properties, where we can define variables that we can access afterwards: =Audi =Red =150 private String brand; private String color; private int power; In this example, the values for the variables are being read from the application.properties file and assigned to them during bean creation. Most of the time, we'd use this approach to inject configuration values from the application.properties file into beans. Default Value Default values are used as a "fallback" if the property we wish to inject isn't defined or missing: private String type; In the above example, because we don't have any car.type property in application.properties, Spring will assign Sedan to the type variable as the default value. If the car.type property gets inserted into the properties file, the new value will be used instead. System Variables We can also access system variables which are stored as properties by the Spring application at start: private String userName; private int numberOfProcessors; private String java; The variables can be called with different naming conventions - Spring searches for us and assigns the correct value. Global Method Value Because @Value is processed by the BeanPostProcessor class, it will be invoked when Spring is building the Spring context by instantiating configuration files and beans. This means that by having @Value on a method, all the arguments will be mapped with the value provided to the annotation: public CarData setCarData(String color, String brand) { carData.setCarColor(color); carData.setCarBrand(brand); } If we are printing carData.getCarColor() and carData.getCarBrand() we will get the car.brand value both times, because like we said, all arguments will be mapped with the value provided. Parameter Method Value We can fix this by using @Value directly on the method parameter: public CarData setCarData(String color, String brand) { carData.setCarColor(color); carData.setCarBrand(brand); } Now, if we are printing the values from the carData object - the color field will have the car.color value, because we provided the value to the parameter of the method itself. Spring Expression Language (SpEL) The Spring Expression Language (SpEL) is an expression language which serves as the foundation for expression evaluation within the Spring portfolio. Basically, when using SpEL together with the @Value annotation, we are just changing the way we tell Spring what we need. Let's take a closer look: private String userName; Check out our hands-on, practical guide to learning Git, with best-practices, industry-accepted standards, and included cheat sheet. Stop Googling Git commands and actually learn it!This is how you'd inject a specific system property. On the other hand, we could inject all properties by: private Map properties; We know now, that we can use the @Value annotation for methods as a global value or as a parameter value. Since constructors are essentially methods, we can use the annotation in constructors as well: public Driver(String name, String location) { this.name = name; this.location = location; } Injecting into Maps With SpEL, we can do some other fairly interesting things when coupled with the @Value annotation. For example, let's make a Map that represents indoor and outdoor hobbies of a student. Each of these can have multiple values: ={indoor: 'reading, drawing', outdoor: 'fishing, hiking, bushcraft'} Now, to inject this, we'll need a Map: private Map hobbies; Injecting into Lists If a property has comma-separated-values, such as a simple list of books, we can use SpEL to interpret it and transform it into a list: =Harry Potter,The Hobbit,Game of Thrones By using the split() method, and splitting for every comma (,), we can inject these values into a list: private List booksRead; Conclusion As soon as you end up working on a real application, you realize that configuration is an important topic and if you are using Spring. There's a big chance you already use or you will have to use the @Value annotation extensively. Understanding this basic functionality is very important because if you don't, you might end up using it totally wrong. We also need to be aware that not everything that looks simple is also very good for long term solutions. For example, we should only use @Value in encapsulated components/services (we can call them configuration services). This way, we will have all our configurations in one place, and that component will only have the responsibility of loading and providing them to other components. 本文主要介绍Spring @Value 注解注入属性值的使用方法的分析，文章通过示例代码非常详细地介绍，对于每个人的学习或工作都有一定的参考学习价值 概述使用方式基于配置文件的注入基于非配置文件的注入注入普通字符串注入操作系统属性注入表达式结果注入其他bean属性注入URL资源 本文小结 概述 本文配置文件为yml文件 在使用spring框架的项目中，@Value是经常使用的注解之一。其功能是将与配置文件中的键对应的值分配给其带注解的属性。在日常使用中，我们常用的功能相对简单。本文使您系统地了解@Value的用法。 @Value 注解可以用来将外部的值动态注入到 Bean 中，在 @Value 注解中，可以使${} 与 #{} ，它们的区别如下： （1）@Value("${}")：可以获取对应属性文件中定义的属性值。 （2）@Value("#{}")：表示 SpEl 表达式通常用来获取 bean 的属性，或者调用 bean 的某个方法。 使用方式 根据注入的内容来源，@ Value属性注入功能可以分为两种：通过配置文件进行属性注入和通过非配置文件进行属性注入。 非配置文件注入的类型如下： 注入普通字符串注入操作系统属性注入表达式结果注入其他bean属性注入URL资源 基于配置文件的注入 首先，让我们看一下配置文件中的数据注入，无论它是默认加载的application.yml还是自定义my.yml文档（需要@PropertySource额外加载）。 application.yml文件配置，获得里面配置的端口号 程序源代码 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value("${server.port}") private String port; @Test public void getPort(){ System.out.println(port); } } 程序结果 自定义yml文件，application-config.yml文件配置，获得里面配置的用户密码值 注意，如果想导入自定义的yml配置文件，应该首先把自定义文件在application.yml文件中进行注册，自定义的yml文件要以application开头，形式为application-fileName 配置信息 测试程序 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value("${user.password}") private String password; @Test public void getPassword(){ System.out.println(password); } } 程序结果 基于配置文件一次注入多个值 配置信息 测试程序 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; import java.util.List; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value("${tools}") private String[] toolArray; @Value("${tools}") private List toolList; @Test public void getTools(){ System.out.println(toolArray); System.out.println(toolList); } } 程序结果 基于非配置文件的注入 在使用示例说明基于非配置文件注入属性的实例之前，让我们看一下SpEl。 Spring Expression Language是Spring表达式语言，可以在运行时查询和操作数据。使用＃{...}作为操作符号，大括号中的所有字符均视为SpEl。 让我们看一下特定实例场景的应用： 注入普通字符串 测试程序 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; import java.util.List; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value("hello world") private String str; @Test public void getValue(){ System.out.println(str); } } 程序结果 注入操作系统属性 可以利用 @Value 注入操作系统属性。 测试程序 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value("#{systemProperties['os.name']}") private String osName; @Test public void getValue(){ System.out.println(osName); } } 程序结果 注入表达式结果 在 @Value 中，允许我们使用表达式，然后自动计算表达式的结果。将结果复制给指定的变量。如下 测试程序 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value("#{ T(java.lang.Math).random() * 1000.0 }") private double randomNumber; @Test public void getValue(){ System.out.println(randomNumber); } } 程序结果 注入其他bean属性 其他Bean package cn.wideth.controller; import org.springframework.beans.factory.annotation.Value; import org.springframework.stereotype.Component; @Component("myBeans") public class OtherBean { @Value("OtherBean的NAME属性") private String name; public String getName() { return name; } public void setName(String name) { this.name = name; } } 测试程序 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value("#{myBeans.name}") private String fromAnotherBean; @Test public void getValue(){ System.out.println(fromAnotherBean); } } 程序结果 注入URL资源 测试程序 package cn.wideth.controller; import cn.wideth.PdaAndIpadApplication; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Value; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.ContextConfiguration; import org.springframework.test.context.junit4.SpringRunner; import java.net.URL; @RunWith(SpringRunner.class) @SpringBootTest() @ContextConfiguration(classes = PdaAndIpadApplication.class) public class ValueController { @Value(") private URL homePage; @Test public void getValue(){ System.out.println(homePage); } } 程序结果 本文小结 本文详细介绍了Spring中的@Value注解的用法。 Image SourceSpring Framework, a prominent platform in the Java ecosystem, provides developers with a plethora of features that simplify the task of creating enterprise-grade applications. One such handy feature is the @Value annotation. This powerful annotation facilitates property injection and allows developers to assign default values, making it a go-to tool in application development.In this tutorial, we will be delving into the usage of the @Value annotation in Spring. Let's get started.What is the @Value Annotation?@Value is a core annotation in Spring that is primarily used for assigning default values to variables and method parameters. It also offers developers the capability to inject values from property files into class members such as fields and method parameters.How to Use the @Value AnnotationTo get started with the @Value annotation, you will need to have a basic Spring Boot project set up. You can easily set one up using the Spring Initializr.Injecting Values from Properties FilesOne of the most common uses for @Value is injecting values from properties files. Let's take a look at how this works.First, let's create a application.properties file in your src/main/resources directory.app.name=MyAppapp.description=This is a demo appNext, in a Spring @Component or @Service, you can inject these properties using the @Value annotation.import org.springframework.beans.factory.annotation.Value;import org.springframework.stereotype.Component;@Componentpublic class MyAppProperties { @Value("${app.name}") private String appName; @Value("${app.description}") private String appDescription; // getters and setters}In this example, ${app.name} and ${app.description} are SpEL (Spring Expression Language) expressions. @Value can handle these expressions and fetch the corresponding values from the properties file.Assigning Default ValuesAnother handy feature of @Value is its ability to assign default values. If a property isn't found, we can assign a default value to it.@Value("${app.version:1.0}")private String appVersion;In the above example, if app.version isn't defined in our properties file, the appVersion field will default to "1.0".Accessing System Properties@Value also provides an easy way to access system properties. For example, you might want to fetch the java.home system property.@Value("${java.home}")private String javaHome;Using @Value with CollectionsThe @Value annotation can also be used with Collections such as Lists and Maps.@Value("#{'${app.users}'.split(',')}")private List appUsers;@Value("#{${app.user.scores}}")private Map userScores;In the properties file, these might be defined as follows:app.users=john,mary,peterapp.user.scores=john:2000,mary:2500,peter:3000Limitations of @ValueWhile @Value is very useful, it does have a few limitations:Type Safety: @Value is not type-safe. It doesn't provide any compile-time checks; thus, any errors related to property resolution will only occur at runtime.Refactoring: If you refactor your properties (like changing their names), you'll need to manually update all @Value references.Support for complex data structures: While @Value supports Lists and Maps, the configuration can get messy for more complex data structures.These limitations have led to the recommendation of using @ConfigurationProperties for complex use cases, or when type safety and autocompletion are needed.ConclusionWe explored how the @Value annotation can be used in Spring for injecting values from property files, assigning default values, and accessing system properties. We also looked at how @Value can be used with Collections. Remember, while @Value is a powerful tool, it does have its limitations, and it is recommended to use @ConfigurationProperties for more complex use cases.That's it for now. Happy coding!Spring Boot icon by Icons8 @Value is typically used to inject externalized properties: @Component public class MovieRecommender { private final String catalog; public MovieRecommender(@Value("${catalog.name}") String catalog) { this.catalog = catalog; } } @Component class MovieRecommender(@Value("\${catalog.name}") private val catalog: String) With the following configuration: @Configuration @PropertySource("classpath:application.properties") public class AppConfig { } @Configuration @PropertySource("classpath:application.properties") class AppConfig And the following application.properties file: catalog.name=MovieCatalog In that case, the catalog parameter and field will be equal to the MovieCatalog value. A default lenient embedded value resolver is provided by Spring. It will try to resolve the property value and if it cannot be resolved, the property name (for example ${catalog.name}) will be injected as the value. If you want to maintain strict control over nonexistent values, you should declare a PropertySourcesPlaceholderConfigurer bean, as the following example shows: @Configuration public class AppConfig { @Bean public static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer() { return new PropertySourcesPlaceholderConfigurer(); } } @Configuration class AppConfig { @Bean fun propertyPlaceholderConfigurer() = PropertySourcesPlaceholderConfigurer() } When configuring a PropertySourcesPlaceholderConfigurer using JavaConfig, the @Bean method must be static. Using the above configuration ensures Spring initialization failure if any ${} placeholder could not be resolved. It is also possible to use methods like setPlaceholderPrefix(), setPlaceholderSuffix(), setValueSeparator(), or setEscapeCharacter() to customize the placeholder syntax. In addition, the default escape character can be changed or disabled globally by setting the spring.placeholder.escapeCharacter.default property via a JVM system property (or via the SpringProperties mechanism). Spring Boot configures by default a PropertySourcesPlaceholderConfigurer bean that will get properties from application.properties and application.yml files. Built-in converter support provided by Spring allows simple type conversion (to Integer or int for example) to be automatically handled. Multiple comma-separated values can be automatically converted to String array without extra effort. It is possible to provide a default value as following: @Component public class MovieRecommender { private final String catalog; public MovieRecommender(@Value("${catalog.name:defaultCatalog}") String catalog) { this.catalog = catalog; } } @Component class MovieRecommender(@Value("\${catalog.name:defaultCatalog}") private val catalog: String) A Spring BeanPostProcessor uses a ConversionService behind the scenes to handle the process for converting the String value in @Value to the target type. If you want to provide conversion support for your own custom type, you can provide your own ConversionService bean instance as the following example shows: @Configuration public class AppConfig { @Bean public ConversionService conversionService() { DefaultFormattingConversionService conversionService = new DefaultFormattingConversionService(); conversionService.addConverter(new MyCustomConverter()); return conversionService; } } @Configuration class AppConfig { @Bean fun conversionService(): ConversionService { return DefaultFormattingConversionService().apply { addConverter(MyCustomConverter()) } } } When @Value contains a SpEL expression the value will be dynamically computed at runtime as the following example shows: @Component public class MovieRecommender { private final String catalog; public MovieRecommender(@Value("#{systemProperties['user.catalog'] + 'Catalog' }") String catalog) { this.catalog = catalog; } } @Component class MovieRecommender(@Value("#{systemProperties['user.catalog'] + 'Catalog' }") private val catalog: String) SpEL also enables the use of more complex data structures: @Component public class MovieRecommender { private final Map countOfMoviesPerCatalog; public MovieRecommender(@Value("#{{'Thriller': 100, 'Comedy': 300}}") Map countOfMoviesPerCatalog) { this.countOfMoviesPerCatalog = countOfMoviesPerCatalog; } } @Component class MovieRecommender(@Value("#{{'Thriller': 100, 'Comedy': 300}}") private val countOfMoviesPerCatalog: Map) Spring @Value annotation is used to assign default values to variables and method arguments. We can read spring environment variables as well as system variables using @Value annotation. Spring @Value annotation also supports SpEL. Let's look at some of the examples of using @Value annotation. We can assign default value to a class property using @Value annotation. @Value("Default DBConfiguration") private String defaultName; @Value annotation argument can be a string only, but spring tries to convert it to the specified type. Below code will work fine and assign the boolean and integer values to the variable. @Value("true") private boolean defaultBoolean; @Value("10") private int defaultInt; @Value("${APP_NAME_NOT_FOUND}") private String defaultAppName; If the key is not found in the spring environment properties, then the property value will be ${APP_NAME_NOT_FOUND}. We can assign a default value that will get assigned if the key is missing from spring environment properties. @Value("${APP_NAME_NOT_FOUND:Default}") private String defaultAppName; When Spring environment is populated, it reads all the system environment variables and stores it as properties. So we can assign system variables too using @Value annotation. @Value("${java.home}") private String javaHome; @Value("${HOME}") private String homeDir; We can also use Spring Expression Language with @Value annotation. So we can read java home system property using SpEL too. @Value("#{systemProperties['java.home']}") private String javaHome; When the @Value annotation is found on a method, Spring context will invoke it when all the spring configurations and beans are getting loaded. If the method has multiple arguments, then every argument value is mapped from the method annotation. If we want different values for different arguments then we can use @Value annotation directly with the argument. @Value("Test") public void printValues(String s, String v){} //both 's' and 'v' values will be 'Test' @Value("Test") public void printValues(String s, @Value("Data") String v){} // s=Test, v=Data Let's create a simple Spring application where we will use @Value annotation to read properties and assign them to class variables. Create a maven project and add spring core dependencies. org.springframework spring-context 5.0.6.RELEASE Our final project structure will look like below image, we will look each of the components one by one. Create a component class where we will inject variable values through @Value annotation. package com.journaldev.spring; import org.springframework.beans.factory.annotation.Value; public class DBConnection { @Value("${DB_DRIVER_CLASS}") private String driverClass; @Value("${DB_URL}") private String dbURL; @Value("${DB_USERNAME}") private String userName; @Value("${DB_PASSWORD}") private char[] password; public DBConnection() { } public void printDBConfigs() { System.out.println("Driver Class = " + driverClass); System.out.println("DB URL = " + dbURL); System.out.println("User Name = " + userName); // Never do below in production environment :D System.out.println("Password = " + String.valueOf(password)); } } Now we have to create a Configuration class and provide a @Bean method for DBConnection class. package com.journaldev.spring; import org.springframework.beans.factory.annotation.Value; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration; import org.springframework.context.annotation.PropertySource; @Configuration @PropertySource("classpath:db.properties") @PropertySource(value = "classpath:root.properties", ignoreResourceNotFound = true) public class DBConfiguration { @Value("Default DBConfiguration") private String defaultName; @Value("true") private boolean defaultBoolean; @Value("10") private int defaultInt; @Value("${APP_NAME_NOT_FOUND:Default}") private String defaultAppName; // @Value("#{systemProperties['java.home']}") @Value("${java.home}") private String javaHome; @Value("${HOME}") private String homeDir; @Bean public DBConnection getDBConnection() { DBConnection dbConnection = new DBConnection(); return dbConnection; } @Value("Test") public void printValues(String s, @Value("another variable") String v) { System.out.println("Input Argument 1 =" + s); System.out.println("Input Argument 2 =" + v); System.out.println("Home Directory = " + homeDir); System.out.println("Default Configuration Name = " + defaultName); System.out.println("Default App Name = " + defaultAppName); System.out.println("Java Home = " + javaHome); System.out.println("Boolean = " + defaultBoolean); System.out.println("Int = " + defaultInt); } } Here is our main class where we are creating annotation-based spring context. package com.journaldev.spring; import java.sql.SQLException; import org.springframework.context.annotation.AnnotationConfigApplicationContext; public class SpringMainClass { public static void main(String[] args) throws SQLException { AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(); context.scan("com.journaldev.spring"); context.refresh(); System.out.println("Refreshing the spring context"); DBConnection dbConnection = context.getBean(DBConnection.class); dbConnection.printDBConfigs(); // close the spring context context.close(); } } When you will run the class, it will produce following output. Input Argument 1 =Test Input Argument 2 =another variable Home Directory = /Users/pankaj Default Configuration Name = Default DBConfiguration Default App Name = Default Java Home = /Library/Java/JavaVirtualMachines/jdk-10.0.1.jdk/Contents/Home Boolean = true Int = 10 Refreshing the spring context Driver Class = com.mysql.jdbc.Driver DB URL = jdbc:mysql://localhost:3306/Test User Name = journaldev Password = journaldev Notice that Configuration class printValues() is getting called before our context is ready to serve user requests. That's all for Spring @Value annotation example, you can download the example code from our GitHub Repository. Spring @Value Annotation Project Thanks for learning with the DigitalOcean Community. Check out our offerings for compute, storage, networking, and managed databases. Learn more about our productsWhile we believe that this content benefits our community, we have not yet thoroughly reviewed it. If you have any suggestions for improvements, please let us know by clicking the "report an issue" button at the bottom of the tutorial.This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License. Spring Tutorial Introduction to Spring Framework Spring Framework Architecture 10 Reasons to Use Spring Framework in Projects Spring Initializr Difference Between Spring DAO vs Spring ORM vs Spring JDBC Top 10 Most Common Spring Framework Mistakes Spring vs Struts in Java How to Download and Install Spring Tool Suite (Spring Tools 4 for Eclipse) IDE? How to Create and Setup Spring Boot Project in Spring Tool Suite? How to Create a Spring Boot Project with IntelliJ IDEA? How to Create and Setup Spring Boot Project in Eclipse IDE? How to Create a Dynamic Web Project in Eclipse/Spring Tool Suite? How to Run Your First Spring Boot Application in IntelliJ IDEA? How to Run Your First Spring Boot Application in Spring Tool Suite? How to Turn on Code Suggestion in Eclipse or Spring Tool Suite? Spring - Understanding Inversion of Control with Example Spring - BeanFactory Spring - ApplicationContext Spring - Difference Between BeanFactory and ApplicationContext Spring Dependency Injection with Example Spring - Difference Between Inversion of Control and Dependency Injection Spring - Injecting Objects By Constructor Injection Spring - Setter Injection with Map Spring - Dependency Injection with Factory Method Spring - Dependency Injection by Setter Method Spring - Setter Injection with Non-String Map Spring - Constructor Injection with Non-String Map Spring - Constructor Injection with Map Spring - Setter Injection with Dependent Object Spring - Constructor Injection with Dependent Object Spring - Setter Injection with Collection Spring - Setter Injection with Non-String Collection Spring - Constructor Injection with Collection Spring - Injecting Objects by Setter Injection Spring - Injecting Literal Values By Setter Injection Spring - Injecting Literal Values By Constructor Injection Bean Life Cycle in Java Spring Custom Bean Scope in Spring How to Create a Spring Bean in 3 Different Ways? Spring - IoC Container Spring - Autowiring Singleton and Prototype Bean Scopes in Java Spring How to Configure Dispatcher Servlet in web.xml File? Spring - Configure Dispatcher Servlet in Three Different Ways How to Configure Dispatcher Servlet in Just Two Lines of Code in Spring? Spring - When to Use Factory Design Pattern Instead of Dependency Injection How to Create a Simple Spring Boot Application? Spring - init() and destroy() Methods with Example Spring WebApplicationInitializer with Example Spring - Project Modules Spring - Remoting by HTTP Invoker Spring - Expression Language (SpEL) Spring - Variable in SpEL What is Ambiguous Mapping in Spring? Spring - Add New Query Parameters in GET Call Through Configurations Spring - Integrate HornetQ Remoting in Spring Framework Spring - Application Events Spring c-namespace with Example Parse Nested User-Defined Functions using Spring Expression Language (SpEL) Spring - AbstractRoutingDataSource Circular Dependencies in Spring Spring - ResourceLoaderAware with Example Spring Framework Standalone Collections How to Create a Project using Spring and Struts 2? Spring - Perform Update Operation in CRUD How to Transfer Data in Spring using DTO? Spring - Resource Bundle Message Source (i18n) Spring Application Without Any .xml Configuration Spring - BeanPostProcessor Spring and JAXB Integration Spring - Difference Between Dependency Injection and Factory Pattern Spring - REST Pagination Spring - Remoting By Burlap Spring - Remoting By Hessian Spring with Castor Example Spring - REST XML Response Spring - Inheriting Bean Spring - Change DispatcherServlet Context Configuration File Name Spring - JMS Integration Spring - Difference Between RowMapper and ResultSetExtractor Spring with Xstream Spring - RowMapper Interface with Example Spring - util:constant Spring - Static Factory Method Spring - FactoryBean Difference between EJB and Spring Spring Framework Annotations Spring Core Annotations Spring - Stereotype Annotations Spring @Bean Annotation with Example Spring Boot @Controller Annotation with Example Spring @Value Annotation with Example Spring @Configuration Annotation with Example Spring @ComponentScan Annotation with Example Spring @Qualifier Annotation with Example Spring Boot @Service Annotation with Example Spring Boot @Repository Annotation with Example Spring - Required Annotation Spring @Component Annotation with Example Spring @Autowired Annotation Spring - @PostConstruct and @PreDestroy Annotation with Example Java Spring - Using @PropertySource Annotation and Resource Interface Java Spring - Using @Scope Annotation to Set a POJO's Scope Spring @Required Annotation with Example Spring Boot Tutorial Spring MVC Tutorial What is Spring Data JPA? Spring Data JPA - Find Records From MySQL Spring Data JPA - Delete Records From MySQL Spring Data JPA - @Table Annotation Spring Data JPA - Insert Data in MySQL Table Spring Data JPA - Attributes of @Column Annotation with Example Spring Data JPA - @Column Annotation Spring Data JPA - @Id Annotation Introduction to the Spring Data Framework Spring Boot - How to Access Database using Spring Data JPA How to Make a Project Using Spring Boot, MySQL, Spring Data JPA, and Maven? Spring - JDBC Template Spring JDBC Example Spring - SimpleJDBCTemplate with Example Spring - Prepared Statement JDBC Template Spring - NamedParameterJdbcTemplate Spring - Using SQL Scripts with Spring JDBC + JPA + HSQLDB Spring - ResultSetExtractor Spring Hibernate Configuration and Create a Table in Database Hibernate Lifecycle Java - JPA vs Hibernate Spring ORM Example using Hibernate Hibernate - One-to-One Mapping Hibernate - Cache Eviction with Example Hibernate - Cache Expiration Hibernate - Enable and Implement First and Second Level Cache Hibernate - Save Image and Other Types of Values to Database Hibernate - Pagination Hibernate - Different Cascade Types Hibernate Native SQL Query with Example Hibernate - Caching Hibernate - @Embeddable and @Embedded Annotation Hibernate - Eager/Lazy Loading Hibernate - get() and load() Method Hibernate Validator CRUD Operations using Hibernate Hibernate Example without IDE Hibernate - Inheritance Mapping Automatic Table Creation Using Hibernate Hibernate - Batch Processing Hibernate - Component Mapping Hibernate - Mapping List Hibernate - Collection Mapping Hibernate - Bag Mapping Hibernate - Difference Between List and Bag Mapping Hibernate - SortedSet Mapping Hibernate - SortedMap Mapping Hibernate - Native SQL Hibernate - Logging by Log4j using xml File Hibernate - Many-to-One Mapping Hibernate - Logging By Log4j Using Properties File Hibernate - Table Per Concrete Class Using Annotation Hibernate - Table Per Subclass using Annotation Hibernate - Interceptors Hibernate - Many-to-Many Mapping Hibernate - Types of Mapping Hibernate - Criteria Queries Hibernate - Table Per Hierarchy using Annotation Hibernate - Table Per Subclass Example using XML File Hibernate - Table Per Hierarchy using XML File Hibernate - Create POJO Classes Hibernate - Web Application Hibernate - Table Per Concrete Class using XML File Hibernate - Generator Classes Hibernate - SQL Dialects Hibernate - Query Language Hibernate - Difference Between ORM and JDBC Hibernate - Annotations Hibernate Example using XML in Eclipse Hibernate - Create Hibernate Configuration File with the Help of Plugin Hibernate Example using JPA and MySQL Hibernate - One-to-Many Mapping Written by: Arvind Rai, Last updated: December 09, 2023 Follow for latest posts: Subscribe for video tutorials:1. Spring @Value annotation is used at field level for expression-driven dependency injection. 2. @Value is also used at parameter level of a method or a constructor that indicates a default value expression for the affected argument. 3. @Value is also used for dynamic resolution of handler method parameters like in Spring MVC. 4. @Value can be used within classes annotated with @Configuration, @Component and other stereotype annotations. 5. The actual processing of @Value annotation is performed by BeanPostProcessor and so we cannot use @Value within BeanPostProcessor types. 6. @Value uses ${...} and #{...} syntax to write expressions. ${...} is a property placeholder syntax and #{...} is a SpEL syntax. #{...} syntax can also handle property placeholders and a lot more. We read property file using @PropertySource annotated at JavaConfig and resolve dependency injection using @Value annotation. Here we will discuss using @Value in detail. Contents ${...} is a property placeholder syntax. It is used with @Value as following. AppConfig.java @Configuration @PropertySource("classpath:myproject.properties") public class AppConfig { @Value("${cp.user.name}") private String userName; ------ } myproject.properties cp.user.name= Shree Mahesh In the above code @PropertySource reads the property file that contains cp.user.name property key. The value of this property will be injected to userName field by @Value. We can also assign hard-coded value to a field using @Value annotation. @Value("India") private String userCountry; The userCountry field will injected with specified value. #{...} is a SpEL syntax. It handles property placeholders and a lot more. @Value("#{systemProperties['java.home']}") private String mySystemVal; systemProperties provides system property values such as file.separator, java.home, os.name etc. In the above code @Value will inject the value of java.home to mySystemVal field. To specify default value for @Value, we need to use colon (:) with ${...} syntax and Elvis Operator (?:) with #{...} syntax to write expression. 1. We need to use colon (:) to specify default value with ${...} syntax. Suppose we have a property file myproject.properties. In our JavaConfig, we read property file with @PropertySource annotation. AppConfig.java @Configuration @PropertySource("classpath:myproject.properties") public class AppConfig { @Value("${cp.user.name:Shree Mahesh}") private String userName; ------ } When cp.user.role property is not defined in property file, the default value Shree Mahesh will be injected to userName field. To specify default value with ${...} syntax, we need not to enclose default value with single quotes (''). If we prefix a space to default value after :, then that will be included in default value. If we suffix a space to default value before }, it will also be included in default value. Look into the following code snippet. a. No space as prefix and suffix. ${cp.user.name:Shree Mahesh} b. Space as prefix. Space will be counted in default value. ${cp.user.name: Shree Mahesh} c. Space as prefix and suffix. Spaces will be counted in default value. ${cp.user.name: Shree Mahesh } 2. If we want to pick default value from property file then we can use ${...} syntax as following. @Value("${cp.user.role:${cp.user.role.default}}") private String userRole; When cp.user.role property is not defined in property file, the default value will be injected to userRole picking it from cp.user.role.default property. 3. To specify default value with #{...} SpEL syntax, we need to use Elvis Operator (?:) as following. @Value("#{systemProperties['unknown_prop_key'] ?: 'Default System Value'}") private String mySystemVal; In the above code snippet, for the demo, we are trying to fetch value from system properties for an unknown key. As the value will be null, so default value will be injected to mySystemVal. 4. To assign a null value as default, we can specify #{null} as following. @Value("${cp.user.country:#{null}}") private String userCountry; When cp.user.country is not defined in property file, then userCountry will be injected with null value. Suppose we have a property key value as true/false in property file. We can inject them using @Value as following. @Value("${cp.user.active}") private Boolean isUserActive; Find code to specify Boolean default value. @Value("${cp.user.active:true}") private Boolean isUserActive; When cp.user.active property is not defined in property file, then true will be injected to isUserActive. Suppose we have a property key with integer value. We can inject them using @Value as following. @Value("${cp.user.age}") private Integer userAge; Find code to specify Integer default value. @Value("${cp.user.age:30}") private Integer userAge; When cp.user.age property is not defined in property file, then 30 will be injected to userAge. Suppose we have following property key in property file with comma separated value. cp.user.skills= Java, Spring, Angular We can inject the comma separated value to Array type field using @Value as following. @Value("${cp.user.skills}") private String[] userSkills; To inject default array, we can use ${...} syntax as following. @Value("${cp.user.skills:Java,Spring,Angular}") private String[] userSkills; When cp.user.skills property is not defined in property file, the default array will be injected to userSkills. In the same way, we can inject primitive datatype integer array as default value, too. @Value("${cp.user.skillids:5,6,7}") private int[] userSkillIds; Suppose we have following property key in property file with comma separated value. cp.user.skills= Java, Spring, Angular We can inject the comma separated value to List type field using @Value with #{...} SpEL syntax and ${...} syntax as following. @Value("#{'${cp.user.skills}'.split(',')}") private List userSkills; Find the code snippet to inject default list value. @Value("#{'${cp.user.skills:Java,HTML,Spring}'.split(',')}") private List userSkills; When cp.user.skills property is not defined in property file, the default list will be injected to userSkills. Suppose we have following key in property file. cp.user.teamMates= {100: 'Krishna', 200: 'Shiva'} We can inject the above key/value to Map type field using @Value as following. @Value("#{${cp.user.teamMates}}") private Map teamMates; To inject default map, we can use #{...} SpEL syntax with ${...} syntax as following. @Value("#{${cp.user.teamMates: {100: 'Krishna', 200: 'Shiva'}}}") private Map teamMates; When cp.user.teamMates property is not defined in property file, the default map will be injected to teamMates. @Value can be used within classes annotated with @Configuration, @Component and other stereotype annotations. Suppose we have a property file with some properties. myproject.properties cp.user.name= Shree Mahesh cp.user.actiontype= Write Find the JavaConfig importing the property file. AppConfig.java @Configuration @PropertySource("classpath:myproject.properties") public class AppConfig { ------ } Now find the component. Here we have injected the value to the component fields using @Value annotation. UserAction.java @Component public class UserAction { @Value("${cp.user.name}") private String userName; @Value("${cp.user.actiontype}") private String actionType; ------ } @Value can be used with constructor parameter and then in constructor injection, Spring will inject value specified by @Value annotation. Suppose we have following properties in property file. myproject.properties cp.company.name= ABC Ltd #cp.company.location= Delhi Find the JavaConfig importing the property file. AppConfig.java @Configuration @PropertySource("classpath:myproject.properties") public class AppConfig { ------ } Find the service that will be instantiated using constructor injection as following. CompanyService.java @Service public class CompanyService { private String compName; private String location; public CompanyService(@Value("${cp.company.name}") String compName, @Value("${cp.company.location:Varanasi}") String location) { this.compName = compName; this.location = location; } ------ } In the above constructor, the first argument will be injected picking the value from property file. If the property key for the second argument is not available, default value will be injected. We can use enum with @Value annotation. A field or constructor parameter can be injected with enum value using @Value annotation. Find a sample enum file. Designation.java public enum Designation { MANAGER, DEVELOPER, TESTER; } Suppose we have a property file with following property. myproject.properties cp.user.designation= DEVELOPER Now inject value to enum field as following. UserAction.java @Component public class UserAction { @Value("${cp.user.designation}") protected Designation designation; ------ } We can use @Value with @ImportResource. Suppose we have an XML configuration that has imported property file using util:properties. app-conf.xml Using #{...} syntax, we can fetch value from userProp as following. AppConfig.java @Configuration @ImportResource("classpath:app-conf.xml") public class AppConfig { @Value("#{userProp['cp.user.name']}") private String userName; @Value("#{userProp['cp.user.role']?: 'Admin'}") private String userRole; ------ } In our JUnit test cases, we can inject test class fields using @Value as following. MySpringAppTest.java @ExtendWith(SpringExtension.class) @ContextConfiguration(classes = AppConfig.class) @TestPropertySource("/myproject.properties") public class MySpringAppTest { @Value("${cp.user.name}") private String userName; @Value("${cp.user.country:India}") private String userCountry; @Test public void userNameTest() throws Exception { assertTrue("Shree Mahesh".equals(userName)); } @Test public void userCountryTest() throws Exception { assertTrue("USA".equals(userCountry)); } } Find the project structure. Now find the complete code. myproject.properties cp.user.name= Shree Mahesh cp.user.country= USA cp.user.role.default= Contributor cp.user.role= Moderator cp.user.active= true #cp.user.age= 50 cp.user.skills= Java, Spring, Angular cp.user.teamMates= {100: 'Krishna', 200: 'Shiva'} cp.company.name= ABC Ltd cp.company.location= Delhi cp.user.actiontype= Write cp.user.designation= DEVELOPER AppConfig.java package com.concretepage; import java.util.List; import java.util.Map; import org.springframework.beans.factory.annotation.Value; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.ComponentScan; import org.springframework.context.annotation.Configuration; import org.springframework.context.annotation.ImportResource; import org.springframework.context.annotation.PropertySource; @Configuration @PropertySource("classpath:myproject.properties") @ComponentScan("com.concretepage") //@ImportResource("classpath:app-conf.xml") public class AppConfig { @Value("${cp.user.name}") private String userName; // @Value("#{userProp['cp.user.name']}") // private String userName; // @Value("#{userProp['cp.user.role']?: 'Admin'}") // private String userRole; @Value("${cp.user.role:Admin}") private String userRole; @Value("India") private String userCountry; @Value("${cp.user.active:false}") private Boolean isUserActive; @Value("${cp.user.age:30}") private Integer userAge; @Value("#{systemProperties['java.home']}") private String mySystemVal; @Value("#{'${cp.user.skills:Java,HTML,Spring}'.split(',')}") private List userSkills; @Value("#{${cp.user.teamMates: {300: 'Mahesh', 400: 'Surya'}}}") private Map teamMates; @Bean public User getUser() { User user = new User(); user.setUserName(userName); user.setUserRole(userRole); user.setUserCountry(userCountry); user.setUserActive(isUserActive); user.setUserAge(userAge); user.setMySystemVal(mySystemVal); user.setUserSkills(userSkills); user.setTeamMates(teamMates); return user; } } User.java package com.concretepage; import java.util.List; import java.util.Map; public class User {

private String userName; private String userRole; private String userCountry; private Boolean userActive; private Integer userAge; private String mySystemVal; private List userSkills; private Map teamMates; //Setters and Getters } Designation.java package com.concretepage; public enum Designation { MANAGER, DEVELOPER, TESTER; } UserAction.java package com.concretepage; import org.springframework.beans.factory.annotation.Value; import org.springframework.stereotype.Component; @Component public class UserAction { @Value("${cp.user.name}") private String userName; @Value("${cp.user.actiontype}") private String actionType; @Value("${cp.user.designation}") protected Designation designation; public void show() { System.out.println(userName); System.out.println(actionType); System.out.println(designation); } } CompanyService.java package com.concretepage; import org.springframework.beans.factory.annotation.Value; import org.springframework.stereotype.Service; @Service public class CompanyService { private String compName; private String location; public CompanyService(@Value("${cp.company.name}") String compName, @Value("${cp.company.location:Varanasi}") String location) { this.compName = compName; this.location = location; } public void showCompanyDetails() { System.out.println("Company Name: "+ compName); System.out.println("Company Location: "+ location); } } MySpringApp.java package com.concretepage; import org.springframework.context.annotation.AnnotationConfigApplicationContext; public class MySpringApp { public static void main(String[] args) { AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(); ctx.scan("com.concretepage"); ctx.refresh(); User user = ctx.getBean(User.class); System.out.println("userName: " + user.getUserName()); System.out.println("userRole: " + user.getUserRole()); System.out.println("userCountry: " + user.getUserCountry()); System.out.println("userActive: " + user.getUserActive()); System.out.println("userAge: " + user.getUserAge()); System.out.println("mySystemVal: " + user.getMySystemVal()); System.out.println("userSkills: " + user.getUserSkills()); System.out.println("teamMates: " + user.getTeamMates()); System.out.println("---UserAction---"); UserAction userAction = ctx.getBean(UserAction.class); userAction.show(); System.out.println("---CompanyService---"); CompanyService compService = ctx.getBean(CompanyService.class); compService.showCompanyDetails(); ctx.registerShutdownHook(); ctx.close(); } } Output userName: Shree Mahesh userRole: Moderator userCountry: India userActive: true userAge: 30 mySystemVal: C:\Program Files\Java\jdk-11.0.1 userSkills: [Java, Spring, Angular] teamMates: {100=Krishna, 200=Shiva} ---UserAction--- Shree Mahesh Write DEVELOPER ---CompanyService--- Company Name: ABC Ltd Company Location: Delhi Spring Doc: @Value SUBSCRIBE TO YOUTUBE CHANNELJOIN THE NEWSLETTERSHAREWRITTEN BY In this article, we'll learn how to inject values for Primitives, List, Map, and Date with inline values and from property file using @Value Annotation with examples. We'll also see its usage with Constructor-based & Setter-based Injection, and SpEL. @Value annotation is used for injecting values into fields in a spring-managed beans. The value typically come from:- Inline Values Property Files ((.properties and .yml files)) System Properties Environment Variables Inject Inline values using @Value Annotation Let's look at the quick examples for how to inject inline values for String, Integer, Float, Double, Boolean, and List using @Value annotation. Injecting inline values for Map, Date, LocalDate, LocalDateTime requires SpEL (Spring Expression Language) to use inside @Value annotation. @Configuration public class InlineConfig { @Value("How to use @Value Annotation with inline values") private String title; @Value("30") private Integer duration; @Value("4.5") private Float rating; @Value("1e+10") private Double pageViews; @Value("true") private Boolean isTrending; @Value("Spring, Spring Boot, Annotation") private List tags; // SpEL expression used to initialize a Map @Value("#{{'keyword1': '12', 'keyword2': '44', 'keyword3': '85', 'keyword4': '100'}}") private Map keywordCountMap; // Inject Date with given format using SpEL expression @Value("#{new java.text.SimpleDateFormat('yyyyMMdd').parse('20210530')}") private Date createdDate; // Inject LocalDate with ISO_DATE format using SpEL expression @Value("#{T(java.time.LocalDate).parse('2021-05-31')}") private LocalDate updatedDate; // Inject LocalDateTime with ISO_LOCAL_DATE_TIME format using SpEL expression @Value("#{T(java.time.LocalDateTime).parse('2015-08-04T10:11:30')}") private LocalDateTime lastAccess; } Inject values from Property file using @Value Annotation application.properties course.title = "How to use Spring @Value annotation" course.duration = 30 course.rating = 4.5 course.page_views = 1e+10 course.trending = true We can set the values of primitive fields such as String, int, float, double and boolean from property file as below:- @Configuration public class CourseConfig { @Value("${course.title}") private String title; //How to use Spring @Value annotation @Value("${course.duration}") private int duration; //30 @Value("${course.rating}") private float rating; //4.5 @Value("${course.page_views}") private double pageViews; //1.0E10 @Value("${course.trending}") private boolean isTrending; //true } Concat multiple properties in @Value Annotation Two or more properties can be concatenated from property file like below:- @Configuration public class CourseConfig { @Value("${course.title} (${course.rating})") private String titleAndRating; //How to use Spring @Value annotation (4.5) @Value("${course.title} - ${course.duration} min") private String titleAndDuration; //How to use Spring @Value annotation - 30 min } Inject with Default values using @Value Annotation Default values can be provided for properties that might not be defined using below syntax: @Value("${property: defaultValue}") For example, property course.review is not defined so review field will be initialized with default value i.e. "No Reviews Yet". @Value("${course.review: No Reviews Yet}") private String review; Inject List using @Value Annotation application.properties application.properties course.tags = Java, Spring, Spring Boot, Annotation Any property having comma separated values can be initialized as a List using @Value annotation in latest version of Spring. // Comma separated property values auto initialize a List @Value("${course.tags}") private List tags; Using SpEL expression with List If you are using older version of Spring, then you need to use SpEL expression to initialize a List: @Value("#{'${course.tags}'.split(',')}") private List tags; SpEL expression can also be used to get specific value from comma separated values: @Value("#{'${course.tags}'.split(',')[0]}") private String firstTag; Inject Map using @Value Annotation application.properties course.keyword_count = { 'keyword1': '12', 'keyword2': '44', 'keyword3': '85', 'keyword4': '100'} Note*: the keys and values in the property Map must be in single quotes ' '. We can initialize a Map with above property using SpEL expression inside @Value annotation // SpEL expression used to initialize a Map @Value("#{${course.keyword_count}}") private Map keywordCountMap; Inject Value of specific Key from Map If we need to get the value of a specific key from Map, all we have to do is add the key's name in the expression: We can use any one out of these two ways:- // way 1 - .key @Value("#{${course.keyword_count}.keyword1}") private Integer FirstKeywordCount; // way 2 - ['key'] @Value("#{${course.keyword_count}['keyword2']}") private Integer SecondKeywordCount; If we're not sure whether the Map contains a certain key, we should choose a safer expression that will not throw an exception but set the value to null when the key is not found: @Value("#{${course.keyword_count}['unknownKey']}") private Integer unknownKeywordCount; Inject Map with Default Value We can also set default values for the properties or keys that might not exist: @Value("#{${unknownMap : {key1: '1', key2: '2'}}}") private Map unknownMap; @Value("#{${course.keyword_count}['unknownKey'] ?: 100}") private Integer unknownKeyWithDefaultValue; Inject Filtered Map Entries Map entries can also be filtered before injection. Let's assume we need to get only those entries whose keyword count is greater than 50: @Value("#{${course.keyword_count}.?[value > '50']}") private Map keywordCountMapFiltered; application.properties course.created_date = 20210530 course.updated_date = 2021-05-31 course.last_access = 2015-08-04T10:11:30 There is no direct support to initialize Date, LocalDate, and LocalDateTime fields using @Value annotation. We need to use SpEL (Spring Expression Language) inside @Value annotation to parse the String based property. // Inject Date with given format using SpEL expression @Value("#{new java.text.SimpleDateFormat('yyyyMMdd').parse('${course.created_date}')}") private Date createdDate; // Inject LocalDate with ISO_DATE format using SpEL expression @Value("#{T(java.time.LocalDate).parse('${course.updated_date}')}") private LocalDate updatedDate; // Inject LocalDateTime with ISO_LOCAL_DATE_TIME format using SpEL expression @Value("#{T(java.time.LocalDateTime).parse('${course.last_access}')}") private LocalDateTime lastAccess; Inject System Properties using @Value Annotation We can initialize fields from system properties using @Value annotation in a similar way as they were defined in a property file. Let's see examples:- //UTF-8 @Value("${file.encoding}") private String fileEncoding; //Mac OS X @Value("${os.name}") private String osName; // /Users/ashishkumarlahoti @Value("${user.home}") private String userHome Inject all System Properties We can also use the @Value annotation to inject all current system properties like this:- @Value("#{systemProperties}") private Map systemPropertiesMap; Output systemPropertiesMap: { "os.name" : "Mac OS X", "os.version" : "10.15.7", "user.name" : "ashl", "user.country" : "SG", "user.language" : "en", "user.timezone" : "Asia/Singapore", "user.home" : "/Users/ashl", "user.dir" : "/Users/ashl/IdeaProjects/springboot-examples/springboot-demo", "file.encoding" : "UTF-8", "file.separator" : "/", "line.separator" : "", "path.separator" : ":", "java.version" : "11.0.10", "java.vendor" : "Oracle Corporation", "java.home" : "/Library/Java/JavaVirtualMachines/jdk-11.0.10.jdk/Contents/Home", "catalina.home" : "/private/var/folders/6p/3ztts6bd4xbdl7nkcq6lpcg80000gn/T/tomcat.8080.8199922893426879147" } Inject value of specific System property from Map We can also use SpEL expressions to get the value from System Property Map: // /Library/Java/JavaVirtualMachines/jdk1.8.0_211.jdk/Contents/Home/jre @Value("#{systemProperties['java.home']}") private String javaHome; //Oracle Corporation @Value("#{systemProperties['java.vendor']}") private String javaVendor; Inject System Property with default value We can also initialize some default value if System property is not available: @Value("#{systemProperties['unknown'] ?: 'Default Value'}") private String unknownSystemProperty; Constructor based Injection using @Value Annotation When we use the @Value annotation, we're not limited to a field-based injection. We can use the @Value annotation for constructor-based injection also. Let's see this in practice: values.properties priority = high @Component @PropertySource("classpath:values.properties") public class PriorityProvider { private String priority; @Autowired public PriorityProvider(@Value("${priority:normal}") String priority) { this.priority = priority; } } In the above example, we inject a property priority directly into the constructor of PriorityProvider. Note that we also provide a default value "normal" in case the priority property is not found. Setter based Injection using @Value Annotation Similar to field-based and constructor-based injection, we can also use @Value annotation for setter-based injection. values.properties listOfValues = first, second, third @Component @PropertySource("classpath:values.properties") public class CollectionProvider { private List values = new ArrayList(); @Autowired public void setValues(@Value("#{'${listOfValues}'.split(',')}") List values) { this.values.addAll(values); } } In the code above, we use the SpEL expression to inject a list of values into the setValues method. Inject Method Arguments using @Value Annotation When the @Value annotation is found on a method, Spring context will invoke it when all the spring configurations and beans are getting loaded. If the method has multiple arguments, then every argument value is mapped from the method annotation. If we want different values for different arguments then we can use @Value annotation directly with the argument. @Configuration public class Config { @Value("Test") public void printValues(String a, String b){ System.out.println(a + " & " + b); // Test & Test } @Value("Test") public void printOtherValues(String a, @Value("Another Test") String b){ System.out.println(a + " & " + b); // Test & Another Test } } SpEL (Spring Expression Language) with @Value Annotation We have already seen usage of SpEL with @Value annotation to inject complex values such as List, Map, and Date where Spring boot doesn't provide direct support. SpEL provides us flexibility to transform or parse the property value before injecting. Let's see some more practical use cases:- Inject Scheme, Host or Port from URL application.properties bootstrap.url = @Value("#{new java.net.URI('${bootstrap.url}').getScheme()}") private String scheme; //http @Value("#{new java.net.URI('${bootstrap.url}').getHost()}") private String host; //localhost @Value("#{new java.net.URI('${bootstrap.url}').getPort()}") private Integer port; //8080 Arithmetic Operations @Value("#{((1 + 2^3 - 4) * (5 % 6)) / 7 }") // 3.0 private Double arithmeticOperation; @Value("#{((1 + 2^3 - 4) * (5 mod 6)) div 7 }") // 3.0 private Double anotherArithmeticOperation; @Value("#{'Hello ' + 'World'}") // "Hello World" private String concatString; We can use either - div or \ for DIVIDE operation, mod or % for MODULO operation. The + operator can also be used to concatenate strings. Relational Operations // @Value("#{1 == 1}") true @Value("#{1 eq 1}") // true private boolean equal; //@Value("#{1 != 1}") // false @Value("#{1 ne 1}") // false private boolean notEqual; // @Value("#{1 < 1}") // false @Value("#{1 lt 1}") // false private boolean lessThan; //@Value("#{1 1}") // false @Value("#{1 gt 1}") // false private boolean greaterThan; //@Value("#{1 >= 1}") // true @Value("#{1 ge 1}") // true private boolean greaterThanOrEqual; We can use either - eq or == for EQUAL operation, != or ne for NOT EQUAL operation, < or lt for LESS THAN operation, > or gt for GREATER THAN operation, = or ge for GREATER THAN EQUAL TO operation Logical Operations //@Value("#{250 > 200 && 200 < 4000}") // true @Value("#{250 > 200 and 200 < 4000}") // true private boolean andOperation; //@Value("#{400 > 300 || 150 < 100}") // true @Value("#{400 > 300 or 150 < 100}") // true private boolean orOperation; //@Value("#{!true}") // false @Value("#{not true}") // false private boolean notOperation; We can use either - && or and for AND operation, || or or for OR operation, ! or not for NOT operation. Conditional Operations The ternary operator can be used inside the expression for conditional logic. @Value("#{2 > 1 ? 'a' : 'b'}") // "a" private String ternaryOperator; Most common use case of ternary operator is to check for null and return default value. @Autowired private SomeBean someBean; @Value("#{someBean.someProperty != null ? someBean.someProperty : 'default'}") private String nullCheckUsingTernaryOperator; SpEL also provide support for Elvis operator ?: which is a shorthand of ternary operator for null check. Above example can be written using Elvis operator like this:- @Value("#{someBean.someProperty ?: 'default'}") // Will inject provided string if someProperty is null private String nullCheckUsingElvisOperator; We learnt different ways to inject values using Spring @Value annotation. We also understood the power and flexibility provided by SpEL to use expressions inside @Value annotation. Download the source code for all the examples from github/springboot-config