* **Difference between spring boot and spring**

1. Spring is java ee framework which is used to develop enterprise application

Where as spring boot is used to develop to rest api

1. In spring the most important feature is dependency injection

Where as in spring boot the important feature is the Autoconfiguration

1. Spring helps to create the loosely coupled application

While in spring boot it create stand alone application

1. We have to set the server to run the application

While spring boot provide embedded server such as a tomcat

1. In spring deployment descriptor is necessary

While it is not necessity in case of spring boot

1. It has to write more nu of code

While it reduce the no of line code

1. It does not provide supports for the in-memory databases.

While it provide in-memory supports for the databases like h2.

* **@ComponentScan**

It is used to scan the component added into the project

* **How many ways , can we create the spring boot application**

Using two ways

1. Using spring initializer
2. Using spring starter projects
3. Spring maven projects
4. Spring boot cli

* **How to change the port**

In application.properties file , write server .port

* **Difference between @controller and @restcontroller**

@controller is used to specify the normal class as a Controller class , which informs the application that , this is an controller layer of project

While

@RestController is used to creating a restful controller . it converts the response into the json and xml .

### 15. What does @PathVariable do in Spring MVC? Why it's useful in REST with Spring? ([answer](http://javarevisited.blogspot.sg/2017/10/differences-between-requestparam-and-pathvariable-annotations-spring-mvc.html))

It's one of the useful annotations from Spring MVC which allows you to read values from URI like query parameter. It's particularly useful in case of creating RESTful web service using Spring because in REST resource identifiers are part of URI.

* **What is dependency injection**

The process of injecting an dependent bean object into a targeted bean is called as dependency injection.

# **Constructor Based Dependency Injection**

Constructor Based Dependency Injection is the **most recommended** dependency injection technique in the Spring Framework. This technique involves passing all necessary dependencies for an object as arguments to the constructor. The object is created only after all necessary dependencies have been provided.

public class ProductService {  
 private final ProductRepository productRepository;  
   
 public ProductService(ProductRepository productRepository) {  
 this.productRepository = productRepository;  
 }  
}

In this example, the ProductService class has a dependency on ProductRepository , which is passed as an argument in the constructor..

Besides the finalkeyword is used to declare the variable productRepositoryas a constant. This way, once a value is assigned in the constructor, the variable cannot be reassigned to another object.

The use of the finalkeyword is optional when injecting dependencies through the constructor, but it is a good programming practice because it helps ensure the immutability and consistency of objects. By making a variable final, you avoid accidentally changing its value in other parts of the code.

Therefore, although the use of finalis not mandatory when injecting dependencies through the constructor, its use is strongly recommended whenever possible.

# **Field or Property-Based Dependency Injection**

Property-Based Dependency Injection(@Autowired on a variable) involves annotating a property with the @Autowired annotation. When Spring creates an object that has a property annotated with @Autowired, Spring looks up an instance of the corresponding dependency and assigns it to the property.

Let’s look at an example:

public class ProductService {  
  
 @Autowired  
 private ProductRepository productRepository;  
}

In this example, the ProductServiceclass has a dependency on ProductRepository, which is injected through the productRepositoryproperty annotated with (@Autowired).

# **Setter Based Dependency Injection**

Setter Based Dependency Injection involves annotating a method with the @Autowired annotation. When Spring creates an object that has a method annotated with @*Autowired*, Spring looks up an instance of the corresponding dependency and assigns it to the method parameter.

Let’s look at an example:

public class ProductService {  
 private ProductRepository productRepository;

@Autowired  
 public void setProductRepository(ProductRepository productRepository) {  
 this.productRepository = productRepository;  
 }  
}

In this example, the ProductServiceclass has a dependency on ProductRepository, which is injected through the setProductRepository() method annotated with (@Autowired).

@SpringBootApplication

A single @SpringBootApplication annotation can be used to enable those three features, that is:

@EnableAutoConfiguration: enable [Spring Boot’s auto-configuration mechanism](https://docs.spring.io/spring-boot/reference/using/auto-configuration.html)

@ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/reference/using/structuring-your-code.html))

@SpringBootConfiguration: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring’s standard @Configuration that aids [configuration detection](https://docs.spring.io/spring-boot/reference/testing/spring-boot-applications.html#testing.spring-boot-applications.detecting-configuration) in your integration tests.

The annotation @SpringBootApplication is the combination of three annotations: **@ComponentScan, @EnableAutoConfiguration,** and **@Configuration**. However, we use @SpringBootApplication annotation instead of using @EnableAutoConfiguration.

**@SpringBootApplication=@ComponentScan+@EnableAutoConfiguration+@Configuration**

Ways to override specific auto configuratin

## **Overriding Built-In Configuration**

By default, the URL path of the web service is case sensitive. For**example:**

[**http://localhost:8080/HelloWorld**](http://localhost:8080/HelloWorld)**!=**[**http://localhost:8080/helloworld**](http://localhost:8080/helloworld)

So, we will make them equal to each other by overriding this configuration. To achieve the goal, we will create a new configuration Class that extends WebMvcConfigurationSupport. Then, we’re going to override configurePathMatch method, as shown below:

1

package com.hashimati.io.demo.overrideconfig.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.util.AntPathMatcher;

import org.springframework.web.servlet.config.annotation.PathMatchConfigurer;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Configuration

public class PathConfig extends WebMvcConfigurationSupport {

@Override

protected void configurePathMatch(PathMatchConfigurer configurer) {

configurer.setPathMatcher(new AntPathMatcher(){

{

setCaseSensitive(false);

}

}

);

}

}

After running the application, any service path will not be case sensitive. The source code is available [here](https://github.com/hashimati/Spring-Boot-Overriding-Configuration).

## **1. What Is a Circular Dependency?**

A circular dependency occurs when a bean A depends on another bean B, and the bean B depends on bean A as well:

Bean A → Bean B → Bean A

Spring cannot decide which of the beans should be created first since they depend on one another. In these cases, Spring will raise a *BeanCurrentlyInCreationException* while loading context.

It can happen in Spring when using **constructor injection.** If we use other types of injections, we shouldn’t have this problem.

Ways to resolve:

We should try to redesign the components properly so that their hierarchy is well designed and there is no need for circular dependencies.

### 4.2. Use @Lazy

A simple way to break the cycle is by telling Spring to initialize one of the beans lazily. So, instead of fully initializing the bean, it will create a proxy to inject it into the other bean. The injected bean will only be fully created when it’s first needed.

@Autowired

**public** **CircularDependencyA**(@Lazy CircularDependencyB circB) {

this.circB = circB;

}

### 4.3. Use Setter/Field Injection

* **What is IOC CONTAINER**

It is basically an container which is used to create dependency injected object , hold the memory address , maintain the full object life cycle of dependency injected object.

@Autowired is a Spring Framework annotation used for dependency injection. It allows Spring to automatically inject dependencies into the components of your application. This annotation can be applied to constructors, methods, and fields.

* **Why the request dispatcher is being used**

For calling one servlet from another, we use request dispatcher

**Spring Actuator**

Its an additional feature of spring that helps you to monitor and manage an application when we push it to production . in actuator , it includes auditing , health , cpu usage , http hits and metric gathering and many more that are applied on a application.

**Dev Tools**

It increase the [productivity of developer.using dev tools , its don’t require ,we don’t require to deploy our application,every time we make the changes.

**How to deal with exception in spring boot**

We can use @ControllerAdvice annotation to handle exception globally

And to deal with specific exception and send customized response , we need to use @ExceptionHandler annotation.

**What are Actuator provided endpoints used for monitoring spring boot application**

HTTPS and JMX endpoints we use to manage and monitor the application

In Spring, a bean's scope defines the lifecycle and visibility of that bean within the Spring container. Different scopes determine how many instances of a bean are created and how long they live. Spring provides several standard scopes for beans, which can be specified using the @Scope annotation or in the XML configuration file.

· **Singleton**: One instance per Spring container (default scope).

· **Prototype**: A new instance for each request.

· **Request**: One instance per HTTP request.

· **Session**: One instance per HTTP session.

· **Application**: One instance per ServletContext.

· **WebSocket**: One instance per WebSocket session.

Starters in spring boot :

spring-boot-starter-web

pring-boot-starter-data-jpa

starter-security

Starter test , starter actuator

Bean In Spring, the @Bean annotation is used to indicate that a method produces a bean to be managed by the Spring container. This annotation is a core part of Spring's Java-based configuration and is used within @Configuration classes. The @Bean annotation tells Spring that the returned object from the method should be registered as a bean within the application context.

### When to Use @Qualifier

When you have more than one bean of the same type and you need to inject a specific one, @Qualifier helps to indicate which bean should be chosen. This is particularly useful in cases where different implementations of an interface are defined as beans.

@Autowired @Qualifier("fooFormatter") **private** Formatter formatter;

Primary vs qualifier

## 1. Spring @Primary annotation – bean preference

When there are multiple beans of the same type, @Primary annotation can be used to give a higher preference to a specific bean.

Spring’s **@Qualifier** annotation. Irrespective of whether any of the beans is annotated with @Primary or not, the @Qualifier annotation will help you select the specific bean you want to inject.

Here are some common HTTP status codes you might use in a Spring application:

* **200 OK**: The request has succeeded.
* **201 Created**: The request has been fulfilled and resulted in a new resource being created.
* **204 No Content**: The server successfully processed the request, but is not returning any content.
* **400 Bad Request**: The server cannot or will not process the request due to a client error.
* **401 Unauthorized**: The request requires user authentication.
* **403 Forbidden**: The server understood the request, but refuses to authorize it.
* **404 Not Found**: The server can't find the requested resource.
* **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.

Mode of autowires

Spring supports the following autowiring modes:

* + **no**: It’s the default autowiring mode. It means

no autowiring.

* + **byName**: The byName

 mode injects the object dependency according to name of the bean. In such case, property name and bean name should be same. It internally calls setter method.

* + **byType**: The byType

 mode injects the object dependency according to type. So it can have different property name and bean name. It internally calls setter method.

* + **constructor**: The constructor

 mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

* + **autodetect**: In this mode, Spring first tries to autowire by

constructor

. If this fails, it tries to autowire by using

byType

## Generated Identifiers

If we want to automatically generate the primary key value, **we can add the *@GeneratedValue* annotation.**

This can use four generation types: AUTO, IDENTITY, SEQUENCE and TABLE.

If we don’t explicitly specify a value, the generation type defaults to AUTO.

This type of generation relies on the *IdentityGenerator*, which expects values generated by an *identity* column in the database. This means they are auto-incremented.

To use this generation type, we only need to set the *strategy* parameter:

@Entity

**public** **class** **Student** {

@Id

@GeneratedValue (strategy = GenerationType.IDENTITY)

**private** **long** studentId;

# How to Access Values From application.properties in Spring Boot?

To access these properties in Spring components, we use annotations like @Value or the Environment object provided by Spring.

Open the **application.properties** file and add the following properties:

spring.application.name=spring-value-demo

@RestController

**public** **class** **ExampleController** {

@Value("${app.title}") *// Injecting the value of app.greeting from application.properties*

**private** String greeting;

@GetMapping("/greet") **public** String greet() { **return** greeting; *// Return the value of app.greeting*

}

}

Spring boot bydefaults starts on 8080 port

role of annotation in spring boot

Its provide data about program , used to provide suppliment information about program.its does not have direct effect on operation.

what happen if we dont give @repository to repository level class in spring boot

Its not neccesarry to put @Repository annotation on interface which extends jpaReposotory, spring automatically recognize it.

The purpose of @NoRepositoryBean is to prevent spring from treating that specific interface as a repositorry by itself.

Auto-configuration? , how does it achieve in spring boot. example of it

Whatever we add the dependency then spring boot automatically performs the auto-configuration.

How spring boot auto configuration works?

Its just have to add dependency , rest all configuration will be taken care by spring boot..

When we add starter jars , then it pull all the jars.and among that jars ,one jar is of auto-configuration.

In that jar contain meta/inf/ spring.factories, inside spring.factories all the configuration keeps mentioned.and all are should be enable or disable based on a some condition.conditin like it checks/scans classpath then if its found then and only loads the auto-configuration.

Spring boot autoconfigure jar -->meta-inf --> spring.factories.

ways to improve performance of spring boot

ChatGPT

Improving the performance of a Spring Boot application can be achieved through several strategies. Here are some effective methods:

### 1. Optimize Database Access

* **Connection Pooling**: Use a connection pool like HikariCP to manage database connections efficiently.
* **Indexes**: Ensure that your database is properly indexed to speed up queries.
* **Batch Processing**: Use batch processing for bulk inserts or updates to reduce the number of database calls.
* **Lazy Loading**: Use lazy loading for relationships to avoid unnecessary data fetching.

### 2. Caching

* **Second-Level Cache**: Use Hibernate's second-level cache to reduce the number of database reads.
* **HTTP Caching**: Implement HTTP caching headers to reduce the load on your application.
* **In-Memory Caches**: Use in-memory caches like Ehcache, Redis, or Caffeine for frequently accessed data.

### 3. Optimize Spring Boot Configuration

* **Profile-Specific Properties**: Use profile-specific properties to optimize configuration for different environments (development, production).
* **Actuator Endpoints**: Monitor and optimize application performance using Spring Boot Actuator endpoints.

### 4. Use Asynchronous Processing

* **@Async**: Use Spring’s @Async annotation for methods that can be executed asynchronously.
* **Executor Configuration**: Properly configure executors to handle asynchronous tasks efficiently.

### 5. Optimize Application Startup

* **Lazy Initialization**: Enable lazy initialization of beans to reduce startup time.
* **Exclude Unused Auto-Configuration**: Exclude auto-configurations that are not used by your application.

to scale spring boot application , to handle high traffic what stratefy u will use?

To scale a Spring Boot application to handle high traffic, you can use a combination of horizontal and vertical scaling strategies, along with optimizing the application and infrastructure. Here are the key strategies:

### 1. Horizontal Scaling

#### Load Balancing

* **Use Load Balancers**: Distribute incoming traffic across multiple instances of your application using load balancers like NGINX, HAProxy, or cloud-based load balancers (e.g., AWS Elastic Load Balancer, Google Cloud Load Balancer).
* **DNS Load Balancing**: Use DNS-based load balancing for geo-distribution.

#### Microservices Architecture

* **Microservices**: Break down the application into smaller, loosely coupled services that can be deployed and scaled independently.
* **Service Discovery**: Use service discovery tools like Netflix Eureka, Consul, or Kubernetes for dynamic discovery of microservices.

### 2. Vertical Scaling

* **Increase Resources**: Scale up your servers by adding more CPU, memory, and storage.
* **Optimized Instances**: Use optimized instance types from cloud providers that are designed for high-performance applications.

### 3. Containerization and Orchestration

#### Docker

* **Dockerize the Application**: Package the application into Docker containers to ensure consistent deployment and easy scalability.

#### Kubernetes

* **Kubernetes**: Use Kubernetes for orchestration, which provides automated deployment, scaling, and management of containerized applications.
* **Helm**: Use Helm charts for managing Kubernetes applications.

### 4. Caching

* **In-Memory Caching**: Use in-memory caching solutions like Redis or Memcached to store frequently accessed data.
* **HTTP Caching**: Implement HTTP caching headers to cache responses at the client side and intermediate proxies.

### 5. Database Scaling

* **Read Replicas**: Use read replicas to offload read traffic from the primary database.
* **Sharding**: Implement database sharding to distribute data across multiple databases.
* **Database Clustering**: Use clustering solutions for databases to improve performance and availability.

### 6. Asynchronous Processing

* **Message Queues**: Use message queuing systems like RabbitMQ, Apache Kafka, or AWS SQS to handle background tasks and offload processing.
* **Asynchronous Requests**: Use asynchronous methods in Spring Boot (@Async) to handle long-running tasks without blocking the main thread.

### 7. Content Delivery Network (CDN)

* **CDN**: Use a CDN to cache and serve static resources, reducing the load on your servers and improving response times.

how would you modify the spring boot application , to make it server less?

ChatGPT

To modify a Spring Boot application to make it serverless, you can deploy it to a serverless platform such as AWS Lambda, Azure Functions, or Google Cloud Functions. Here are the steps and considerations for making a Spring Boot application serverless:

### 1. Choose a Serverless Platform

Select a serverless platform that best suits your needs. The most common choices are:

* **AWS Lambda**
* **Azure Functions**
* **Google Cloud Functions**

### 2. Optimize the Application for Serverless

Serverless environments are stateless, and they scale automatically based on demand. Ensure your Spring Boot application is optimized for such an environment:

* **Stateless Design**: Ensure your application does not rely on in-memory state.
* **Fast Start-Up**: Minimize the application’s start-up time by using techniques such as lazy initialization and removing unnecessary components.

### 3. Use a Lightweight Spring Boot Configuration

Minimize the use of dependencies and configurations that are not required for serverless functions:

* **Remove Web Server**: A full-fledged web server like Tomcat or Jetty may not be necessary. You can use a minimal configuration if your function only handles specific tasks.

How to fetch the value from env.propertries

Using @Value

@Value(“#{systemProperties[‘app.environment’]}”)

Private String anm;

Using environment variable

@Autowired

**private** Environment ev;

How to connect 2 db

1. create prperties in app.propertiies forall db we wanna use.
2. Then create entities, repos, andconfig for both two db
3. Keep entities in different packagres
4. Then configure the configure class of both diferent class using @enablejparepositories
5. And create method with @Bean and @primary and inside it perform datasource configuration

how to disable autoconffiguration forspecific class in ssprinmnngng boot

@EnableAutoConfiguration annotation with the exclude or excludeName attributes. Here are a couple of ways to achieve this:

**Using** exclude **Attribute**: You can exclude specific auto-configuration classes by listing them in the exclude attribute of the @EnableAutoConfiguration or @SpringBootApplication annotation

import org.springframework.boot.autoconfigure.SpringBootApplication;import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

**Using** excludeName **Attribute**: If you prefer to exclude auto-configuration classes by their fully qualified class names, you can use the excludeName attribute.

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(excludeName = {"org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration"})public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

**Using** excludeName **Attribute**: If you prefer to exclude auto-configuration classes by their fully qualified class names, you can use the excludeName attribute.

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(excludeName = {"org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration"})public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

**Using** spring.autoconfigure.exclude **Property**: Another way to exclude auto-configuration classes is by setting the spring.autoconfigure.exclude property in your application.properties or application.yml file.

**application.properties**:

properties

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

Spring Boot provides a flexible logging system that you can configure to set different logging levels for various packages or classes. By default, Spring Boot uses Logback for logging, but it also supports other logging frameworks like Log4j2 and Java Util Logging (JUL).

### Configuring Logging Levels in application.properties

You can set logging levels for different packages or classes in the application.properties file. Here are the available logging levels:

* TRACE
* DEBUG
* INFO
* WARN
* ERROR
* FATAL
* OFF

when we use query param and path variable in spring boot

Use @RequestParam for accessing query parameters when the parameters are optional or when dealing with a large number of parameters. Use @PathVariable when the parameters are mandatory and part of the URL path, especially in RESTful APIs where the URL structure reflects the resource hierarchy.

How to set different profiles in spring boot

1. create three different application.properties and assign db properties
2. Then create configuration file and inside create bean using @bean and @profile
3. Then in application.properties put spring.profile.active = dev

### Steps to Create a Custom Annotation in Spring Boot

**Define the Custom Annotation** To create a custom annotation, you need to use @interface in Java. Specify the meta-annotations (like @Target, @Retention, etc.) that define where and how the annotation can be used.

**Add Necessary Meta-Annotations** You can control the annotation’s behavior using meta-annotations like @Target, @Retention, etc.

**Apply the Annotation** Once the annotation is defined, you can use it on classes, methods, or fields as per its scope.

@Target(ElementType.METHOD) // The annotation can be applied to methods @Retention(RetentionPolicy.RUNTIME) // The annotation will be available at runtime public @interface LogExecutionTime { }

#### 2. Create an Aspect for the Annotation

Using Spring AOP, we can define an aspect that will apply cross-cutting concerns (like logging) to the methods annotated with @LogExecutionTime.

@Aspect

@Component

public class LogExecutionTimeAspect {

@Around("@annotation(com.example.annotations.LogExecutionTime)")

// Apply the aspect around methods annotated with @LogExecutionTime public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {

long startTime = System.currentTimeMillis();

Object proceed = joinPoint.proceed(); // Execute the method long executionTime = System.currentTimeMillis() - startTime; System.out.println(joinPoint.getSignature() + " executed in " + executionTime + "ms");

return proceed;

}

}

#### 3. Apply the Annotation to a Method

Now that we have the custom annotation and the aspect ready, we can apply the annotation to any method that we want to log the execution time for.

@LogExecutionTime

public void serve() throws InterruptedException {

Thread.sleep(2000); // Simulate a long-running task System.out.println("Method serve() executed");

}

### Steps Summary:

1. **Define the custom annotation** using @interface and meta-annotations.
2. **Create an aspect** using Spring AOP to provide functionality for the annotation.
3. **Apply the annotation** to methods, classes, or fields where you want it to take effect.
4. **Enable AspectJ support** in Spring Boot with @EnableAspectJAutoProxy.

application.properties or application.yml,which you prefer and why

YAML is often favored for its human-readable syntax and support for complex data structures, while . properties files are more concise and familiar to developers with a background in Java. Readability and Conciseness:

@Async

In Spring Boot, the @Async annotation can be used to run a method asynchronously, meaning it will run in a separate thread, allowing the main thread to continue processing without waiting for the completion of the asynchronous method.

### Step 1: Enable Async Support

First, you need to enable async support in your Spring Boot application. This can be done by adding the @EnableAsync annotation to one of your configuration classes.

import org.springframework.context.annotation.Configuration;import org.springframework.scheduling.annotation.EnableAsync;

@Configuration

@EnableAsync

public class AppConfig {

}

@Async is an annotation will be on class or method name , we mentioned that this will execute seperately or parallally without waiting for anyone thread.

can we secure actuator endpoints

Yes using security

what is cors in spring boot

Cross-origin resource sharing (CORS) is a mechanism for integrating applications. CORS defines a way for client web applications that are loaded in one domain to interact with resources in a different domain.Cross-origin resource sharing (CORS) is a mechanism for integrating applications. CORS defines a way for client web applications that are loaded in one domain to interact with resources in a different domain.

@CrossOrigin(origins = "http://localhost:9000")

And we can do that in global level like by using webmvcconfigurer

Inside that use alloOrigin

@Bean public

WebMvcConfigurer corsConfigurer() {

return new WebMvcConfigurer() {

@Override public void addCorsMappings(CorsRegistry registry)

{

registry.addMapping("/greeting-javaconfig").allowedOrigins("http://localhost:9000");

}

};

}

[Question 43: What is the difference between a WAR file & an embedded container?](https://www.synergisticit.com/spring-boot-interview-question-part-3/#a210ee87ba5b90add)

The main difference between a WAR file & an embedded system is that an embedded container allows Spring Boot applications to run as a JAR directly from the command prompt without setting up any web server. However, to run a WAR file, you need to set up a web server first like Tomcat, which has a Servlet container, then you need to deploy WAR to run Spring Boot applications.

what is autowiring and its types in spring boot

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

## Autowiring Modes

There are many autowiring modes:

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 1) | no | It is the default autowiring mode. It means no autowiring bydefault. |
| 2) | byName | The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method. |
| 3) | byType | The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method. |
| 4) | constructor | The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters. |

@Inject and @Autowired both annotations are used for autowiring in your application.

 Both annotations fulfill same purpose

Sr. No.Key@Inject@Autowired1  
Basic  
It is part of Java CDI  
It is part of Spring framework  
2  
Required  
It has no required attribute  
It has required attribute  
3  
Default Scope  
Default scope of the autowired beans is Singleton  
Default scope of the inject beans is prototype  
4  
Ambiguity  
In case of ambiguity in beans for injection then @Named qualifier should be added in your code.  
In case of ambiguity in beans for injection then @Qualifer  qualifier should be added in your code.  
5  
Advantage  
It is a part of Java CDI so it is not dependent on any DI framework. It makes your system loosely coupled.  
It makes your application tightly coupled with Spring framework. In the future , if you want to move to another DI framework then you need reconfigure your application.

difference between bean and component

@Bean is used at the method level to explicitly declare individual beans with custom instantiation logic, while @Component is used at the class level to mark classes as Spring-managed components that are automatically discovered during component scanning.

preconstructy , post construct , predestroy post destroy in spring boot

Spring allows us to attach custom actions to [bean creation and destruction](https://www.baeldung.com/running-setup-logic-on-startup-in-spring).

## **.2. @PostConstruct**

**Spring calls the methods annotated with @PostConstruct only once, just after the initialization of bean properties**

@Autowired **private** UserRepository userRepository; @PostConstruct **private** **void** **postConstruct**() {

The above example will first initialize *UserRepository* and then run the *@PostConstruct* method.

## **3. @PreDestroy**

A method annotated with @PreDestroy runs only once, just before Spring removes our bean from the application context.

@Component **public** **class** **UserRepository** {

**private** DbConnection dbConnection;

@PreDestroy **public** **void** **preDestroy**() {

how sprinbg bopt application boostraps

### Summary of the Bootstrap Process:

1. **Run the** main() **method** to start the application.
2. **SpringApplication** creates the application context and initializes the environment.
3. **Auto-configuration** is applied based on classpath contents.
4. **ApplicationContext** is refreshed and Spring-managed beans are initialized.
5. **Embedded server** (if applicable) is started.
6. The application is **ready** to serve requests or execute tasks.

# Spring Boot – Versioning a REST API

**Last Updated :**17 May, 2024

API Versioning is a defined process of making or managing changes to an API. These changes can be made transparently without interrupting the clients.

## REST API Versioning Strategies – Spring Boot

There are several types of strategies to versioning REST API using Spring Boot.

1. URI Path Versioning
2. Query Parameters Versioning
3. Custom Header Versioning
4. Content Negotiation Versioning

### 1. URI Path Versioning

The process of adding the different version numbers like v1,v2, etc. to the API’s URL is known as URI Versioning. It is easy to understand and straightforward. Sometimes it may lead to long URL Paths.

**Example** :

* http://www.your-website.com/v1/data or http://localhost:8080/v1/data
* http://www.your-website.com/v2/data or http://localhost:8080/v2/data

Java

At the end version number we have to send , and we can send fromabove mentionbed way.

Validation in spring boot

Spring Boot provides various mechanisms for validation, including annotations, custom validators, error handling and group validation.

public class UserRegistrationForm {  
@NotBlank(message = "Please provide a username")  
private String username;

1. **@NotNull:** Ensures a field is not null.
2. **@NotBlank:** Enforces non-nullity and requires at least one non-whitespace character.
3. **@NotEmpty:**Guarantees that collections or arrays are not empty.
4. **@Min(value):** Checks if a numeric field is greater than or equal to the specified minimum value.
5. **@Max(value):** Checks if a numeric field is less than or equal to the specified maximum value.
6. **@Size(min, max):** Validates if a string or collection size is within a specific range.
7. **@Pattern(regex):**Verifies if a field matches the provided regular expression.
8. **@Email:** Ensures a field contains a valid email address format.
9. **@Digits(integer, fraction):** Validates that a numeric field has a specified number of integer and fraction digits.
10. **@Past** and **@Future** : Checks that a date or time field is in the past and future respectively.
11. **@AssertTrue** and **@AssertFalse**: Ensures that a boolean field is true. and false respectively.
12. **@CreditCardNumber**: Validates that a field contains a valid credit card number.
13. @Valid: Triggers validation of nested objects or properties.
14. @Validated: Specifies validation groups to be applied at the class or method level.

## **13. @Valid**

When you apply the @Valid annotation to a method parameter, Spring Boot automatically triggers validation for that parameter before the method is invoked. It is placed before the object to indicate that it should be validated. This means that the incoming data for that parameter will be validated against the specified validation rules.

@PostMapping("/create")  
public ResponseEntity<String> createUser(@RequestBody @Valid User user,

[Question 36. How to register a custom auto-configuration?](https://www.synergisticit.com/spring-boot-interview-questions-and-answers-part-2/#509ea8f4952d43d6d)

To register a custom auto-configuration class with Spring Boot, you must have its fully-qualified name listed under the EnableAutoConfiguration key in the META-INF/spring.factories file.

Conditional annmotation in spring boot

The Spring Framework provides a powerful way to control bean creation and when certain beans and configuration loaded based on environment , presence of class , properties or other conditions.

Spring starter parent:

It is a starter project that defines the default configuration of spring based application

Provide default compiler level such as jdk 1.8 and utf source code encoding

Maven life cycle ;

Resource filtering and configuring profile specific files

# Bean life cycle in Java Spring

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

### Difference between BeanFactory and the ApplicationContext

where should @Component use

### 1. Generic Components

Use @Component for generic components that do not fall into the specific categories of service, repository, or controller.

In summary, use @Component for generic components that don't fit into the service, repository, or controller categories, and for utility classes, factory beans, event listeners, and AOP aspects.

Validator in java

@NotNull , NotEmpty , NpootBlank , min and max , email

Precaution while using dev tools

Ensure that development tools are only enabled in non-production environments

### 2. Exclude Dev Tools from Production Builds

Exclude dev tools dependencies from production builds to avoid performance and security issues.

### Key Strategies for Performance Optimization in Spring Data JPA

Below are some approaches, which will help us to optimize the performance.

1. Lazy Loading
2. Pagination
3. Caching
4. Batch Processing

complex queries that runs slowly , how would you optimize in spring boot

### Database Indexing

Ensure that your database is properly indexed. Indexes can drastically improve read performance for queries.

* **Identify Slow Queries**: Use your database's slow query log or performance monitoring tools.
* **Add Indexes**: Create indexes on columns that are frequently used in WHERE clauses, JOIN conditions, and ORDER BY clauses.

### 2. Optimize Queries

Rewrite or refactor your SQL queries to be more efficient.

* **Use Joins Efficiently**: Ensure that JOIN operations are optimal and avoid unnecessary joins.
* **Select Only Necessary Columns**: Avoid SELECT \* and specify only the columns you need.
* **Avoid Subqueries**: Where possible, replace subqueries with JOINs or common table expressions (CTEs).
* **Use Batching**: For operations involving large data sets, consider using batch processing to minimize transaction times.

### 3. Hibernate/JPA Tuning

If you are using Hibernate or JPA, there are additional considerations.

* **Fetch Strategies**: Use appropriate fetch strategies (LAZY vs EAGER). Eager fetching can lead to large result sets and potentially slow queries.
* **Cache**: Enable second-level caching if appropriate for your use case.

### 5. Pagination

Implement pagination for queries that return large result sets.

* **Spring Data JPA Pagination**: Use Pageable interface to handle pagination in Spring Data JPA.

Request param and path variable

@RestController

@RequestMapping("/api/items")

public class ItemController { // Using @RequestParam to extract query parameter

@GetMapping

public String getItemByName(@RequestParam(name = "name") String name) {

return "Item requested: " + name;

}

// Using @PathVariable to extract from URI path

@GetMapping("/{name}")

public String getItemByPath(@PathVariable("name") String name) {

return "Item requested: " + name;

}

}

Caching sprb

It stores the recently used data that reduces the number of database hits as much as possible. In other words, caching is to store data for future reference.

@Schedule

Creating a *@Component* class *Scheduler* which defines the method *scheduleTask()* for scheduling a task using the ***@Scheduled*** annotation.v

*// To trigger the scheduler every one minute*

*// between 19:00 PM to 19:59 PM*

@Scheduled(cron = "0 \* 19 \* \* ?")

**Traditional WAR Deployment in a Servlet Container**

1. Develop a Java EE web application using **Servlets, JSP, or frameworks like Struts/Spring MVC**.
2. Package the application as a **WAR (Web Application Archive)**.
3. Deploy the WAR file into a **Servlet Container** like Apache Tomcat or Jetty.

how to accept xml data as rest in java spring boot

add this depencency   
jackson-dataformat-xml and jaxb runtime

then create model and annotate with @JacksonXmlRootElement

and add this above method decleration

@PostMapping(value = "/create", consumes = MediaType.APPLICATION\_XML\_VALUE, produces = MediaType.APPLICATION\_XML\_VALUE)