*Spring annotation*

*@Component Annotation :*

*Is class level annotation is denote class as component*

*Use := to make bean to manage spring component*

*Component responsible for some other annotation like*

## *@Service*

## *@ repository database*

## *@ Controller*

*org.springframework.stereotype*

*@Service*

1. *To indicate that hold as business logic*
2. *The utility classes can be marked as Service classes.*

*@ repository*

1. *to indicate that they’re dealing with CRUD operations, usually, it’s used with DAO (Data Access Object) or Repository implementations that deal with database tables*

*@ Controller*

1. *@Controller to indicate that they’re front controllers and responsible to handle user requests and return the appropriate response. It is mostly used with REST Web Services.*

***@Autowired Annotation***

1. *@Autowired annotation marks a Constructor, Setter method, Properties and Config() method as to be autowired that is ‘injecting beans'(Objects) at runtime by Spring Dependency*
2. *By declaring beans, you provide metadata to the Spring Container to return the required dependency object at runtime. This is called Spring Bean Autowiring*
3. *all the bean methods are defined in the class with****@configuration****annotation. At runtime, Spring will provide bean definitions by reading those methods.*

***@Entity***

*The Customer class is annotated with****@Entity****annotation and defines getters, setters, and constructors for the fields.*

## ***@Repository***

*@Repository is a Spring annotation that indicates that the decorated class is a repository. A repository is a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects. It is a specialization of the @Component annotation allowing for implementation classes to be autodetected through classpath scanning.*

**Core Annotations**

* **@SpringBootApplication**
  + **Internal Working**: This annotation is a combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan. It marks the main class of a Spring Boot application. Internally, it sets up the application context, enables auto-configuration of beans, and scans for components in the package and its sub-packages.
* **@Configuration**
  + **Internal Working**: Marks a class as a source of bean definitions. When the Spring container sees this annotation, it processes the class as a configuration class, and methods annotated with @Bean are invoked to register beans in the Spring application context.
* **@ComponentScan**
  + **Internal Working**: Directs Spring to scan the specified packages for components, configurations, and services. This annotation helps in locating and registering beans annotated with @Component, @Service, @Repository, and @Controller.
* **@Bean**
  + **Internal Working**: Declares a method that returns an object to be registered as a Spring bean. The Spring container calls this method to create and manage the bean's lifecycle.
* **@Component**
  + **Internal Working**: Indicates that a class is a Spring-managed component. Spring automatically detects and registers it as a bean in the application context during component scanning.
* **@Service**
  + **Internal Working**: Specialization of @Component, used to annotate service classes. It is a marker for service-layer classes, helping with clarity and design, although functionally similar to @Component.
* **@Repository**
  + **Internal Working**: Specialization of @Component, used for DAO (Data Access Object) classes. It adds additional behavior, such as exception translation, converting database-specific exceptions into Spring’s data access exceptions.
* **@Controller**
  + **Internal Working**: Marks a class as a Spring MVC controller. It is used to handle web requests and return views or data. The @Controller annotation integrates with the DispatcherServlet for handling HTTP requests.
* **@RestController**
  + **Internal Working**: A convenience annotation combining @Controller and @ResponseBody. It automatically serializes the return value of methods to JSON or XML, making it suitable for RESTful web services.

**Configuration and Management Annotations**

* **@Autowired**
  + **Internal Working**: Used for automatic dependency injection. Spring resolves and injects the required dependencies into fields, constructors, or setter methods by scanning the application context.
* **@Qualifier**
  + **Internal Working**: Helps resolve ambiguity when multiple beans of the same type exist. Spring uses the specified qualifier to determine which bean to inject.
* **@Value**
  + **Internal Working**: Injects values into fields from property files or environment variables. Spring reads the specified property value and injects it into the annotated field.
* **@Profile**
  + **Internal Working**: Specifies that a bean or configuration should be active only for specific profiles. Spring evaluates the active profiles at runtime and includes or excludes beans based on the profile configuration.
* **@ConditionalOnProperty**
  + **Internal Working**: Conditionally registers a bean based on the presence or value of a specified property. It checks application properties at runtime and only creates the bean if the conditions match.

**Web and Security Annotations**

* **@RequestMapping**
  + **Internal Working**: Maps HTTP requests to handler methods. It uses a combination of URL patterns, HTTP methods, and other attributes to route requests to the appropriate method in the controller.
* **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**
  + **Internal Working**: Specialized versions of @RequestMapping for specific HTTP methods. They provide a more concise way to handle requests corresponding to GET, POST, PUT, and DELETE operations.
* **@PathVariable**
  + **Internal Working**: Binds a method parameter to a URI template variable. Spring extracts the value from the URL and injects it into the annotated method parameter.
* **@RequestParam**
  + **Internal Working**: Binds request parameters to method parameters. Spring retrieves the parameters from the query string or form data and injects them into the method.
* **@RequestBody**
  + **Internal Working**: Binds the HTTP request body to a method parameter. Spring deserializes the request body into the specified Java object, usually JSON or XML.
* **@ResponseBody**
  + **Internal Working**: Indicates that the return value of a method should be written directly to the HTTP response body. Spring serializes the return value into the appropriate format (e.g., JSON) for the response.
* **@ExceptionHandler**
  + **Internal Working**: Handles exceptions thrown by handler methods. Spring invokes the annotated method when an exception of the specified type is thrown, allowing for custom error handling logic.

**Data Management Annotations**

* **@Entity**
  + **Internal Working**: Marks a class as a JPA entity. Spring scans and registers the class as an entity in the persistence context, mapping it to a database table.
* **@Table**
  + **Internal Working**: Specifies the table name in the database that an entity maps to. Spring uses this annotation to determine the table name during database operations.
* **@Id**
  + **Internal Working**: Defines the primary key of an entity. Spring JPA uses this field to uniquely identify each entity instance in the database.
* **@GeneratedValue**
  + **Internal Working**: Specifies the strategy for generating primary key values. It can use strategies such as auto-increment, sequence, or UUID.
* **@Column**
  + **Internal Working**: Maps a field to a column in the database table. It allows customization of column properties such as name, length, and nullability.
* **@RepositoryRestResource**
  + **Internal Working**: Customizes the RESTful endpoints exposed by a Spring Data REST repository. It provides configuration options for resource paths and repository behavior.

**Testing Annotations**

* **@SpringBootTest**
  + **Internal Working**: Loads the full application context for integration testing. It sets up the application environment and provides access to all beans and components for comprehensive testing.
* **@DataJpaTest**
  + **Internal Working**: Configures an in-memory database and scans for JPA repositories. It provides a focused environment for testing JPA-based persistence logic without starting the full application context.
* **@WebMvcTest**
  + **Internal Working**: Configures only the web layer of the application for testing. It includes controllers and related components but does not load the entire application context or services.
* **@MockBean**
  + **Internal Working**: Adds Mockito mocks as beans in the Spring application context. It allows for mocking dependencies in unit tests by replacing actual beans with mocks.

***Difference Table***

|  |  |
| --- | --- |
| ***CrudRepository*** | ***JpaRepository*** |
| *It is a base interface and extends Repository Interface.* | *It extends PagingAndSortingRepository that extends CrudRepository.* |
| *It contains methods for CRUD operations. For example save(), saveAll(), findById(), findAll(), etc.* | *It contains the full API of CrudRepository and PagingAndSortingRepository. For example, it contains flush(), saveAndFlush(), saveAllAndFlush(), deleteInBatch(), etc along with the methods that are available in CrudRepository.* |
| *It doesn’t provide methods for implementing pagination and sorting* | *It provides all the methods for which are useful for implementing pagination.* |
| *It works as a marker interface.* | *It extends both CrudRepository and PagingAndSortingRepository.* |
| *To perform CRUD operations, define repository extending CrudRepository.* | *To perform CRUD as well as batch operations, define repository extends JpaRepository.* |
| ***Syntax:***  *public interface CrudRepository<T, ID> extends Repository<T, ID>* | ***Syntax:***  *public interface JpaRepository<T,ID> extends PagingAndSortingRepository<T,ID>, QueryByExampleExecutor<T>* |

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*@Requstmapping*

*Its use to map web request , its has many optional element like consumer , header ,method ,name params ,path produces and value we use class as well as method*

*Spring currently support five type of inbuilt annotation for handling incoming request which are GET ,POST ,DELETE PUT PATCH ,POST*

## Five types of Request in java

* *HTTP GET Request: This type of request is used to retrieve data from a server. It is commonly used to fetch resources or retrieve information from the server.*
* *HTTP POST Request: This type of request is used to send data to the server to create or update a resource. It is commonly used for form submissions, data uploads, or API requests that modify server-side data.*
* *HTTP PUT Request: This type of request is used to send data to the server to update or replace an existing resource. It is commonly used for updating or editing existing data on the server.*
* *HTTP DELETE Request: This type of request is used to delete a specified resource on the server. It is commonly used to remove data or resources from the server.*
* *HTTP PATCH Request: This type of request is used to send a partial update to a resource on the server. It is commonly used when only a portion of the resource needs to be modified.*

# *what :*

# *paging refers to the process of dividing memory into fixed-size blocks called pages. However, Java abstracts away the low-level details of paging, as memory management is handled by the Java Virtual Machine (JVM).*

# *How:*

# *The JVM automatically manages memory through garbage collection and virtual memory techniques, so as a Java developer, you don't need to directly deal with paging. The JVM takes care of memory allocation and deallocation, allowing you to focus on writing your code.*

## *exception handling*

# *what*

*Exception handling is a programming concept that deals with handling and responding to unexpected or exceptional situations that occur during the execution of a program. When an exceptional situation, known as an exception, arises, it disrupts the normal flow of the program and can cause errors or undesired behavior.*

***How***

* *@ControllerAdvice: Spring Boot allows you to define a global exception handler using the @ControllerAdvice annotation. You can create a class annotated with @ControllerAdvice and define methods annotated with @ExceptionHandler to handle specific exceptions or groups of exceptions.*
* *@ExceptionHandler: By using the @ExceptionHandler annotation, you can define methods within your controllers to handle specific exceptions thrown by those controllers. These methods can return custom error messages, redirect to error pages, or perform any other necessary actions.*
* *Custom Exception Classes: You can create your custom exception classes by extending the base RuntimeException or Exception classes provided by Java. These custom exceptions can be thrown within your application's code, and you can handle them using the @ExceptionHandler methods mentioned earlier.*
* *@ResponseStatus: The @ResponseStatus annotation can be used in conjunction with custom exceptions to define the HTTP status code returned when the exception is thrown. By setting the appropriate status code, you can provide meaningful error responses to clients.*
* *Default Error Handling: Spring Boot automatically provides a default error handling mechanism that returns a standard error response when an unhandled exception occurs. This response includes details such as the error message, stack trace, and the HTTP status code.*

## Two type of exception in java

Checked Exceptions: *These are exceptions that are checked by the Java compiler at compile-time. Methods that can throw checked exceptions must declare them using the throws keyword. Examples of checked exceptions include IOException, SQLException, and FileNotFoundException. Checked exceptions represent exceptional conditions that can be reasonably expected and handled by the calling code.*

Unchecked Exceptions: *Also known as runtime exceptions, these exceptions do not need to be declared explicitly in the method signature. They occur at runtime and are not checked by the compiler. Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException. Unchecked exceptions typically indicate programming errors or unexpected conditions that should be fixed in the code.*

## *Why java is not 100% object oriented programing language*

## *What :*

*In an object-oriented programming language, everything is treated as an object, including basic data types.*

## *why*

*because of some reason java is not 100 % object-oriented because*

* *it includes primitive data types, such as int, float, and boolean, which are not objects and do not have the same capabilities as objects.*
* *it supports procedural programming constructs, such as static methods and variables, which do not rely on objects.*

## *Why pointers are not used in java*

## *What is pointer*

## *Pointers are variables that store memory addresses. They are used in low-level languages like C and C++ to directly manipulate memory and access data structures. Pointers allow for efficient memory management and indirect modification of data*

## *Why not allow*

*Because of some security reasons java did not support pointer*

* *They are unsafe*
* *Increases complex city and java is known for simplicity*
* *Since JVM is responsible for the implementation of memory allocation this avoids direct access to memory by the user*
* *Java references allow indirect manipulation of objects, similar to pointers, but they are strongly typed and automatically managed by garbage collection, eliminating manual memory management.*

## *JSON web token | JWT*

## *What :*

*A JSON web token(JWT) is JSON Object which is used to securely transfer information over the web(between two parties).*

*why:*

* *It can be used for an authentication system and can also be used for information exchange.*
* *The token is mainly Captain* ***header, payload, signature.*** *These three parts are separated by dots(.).*
* *JWT defines the structure of information we are sending from one party to the another, and it comes in two forms –****Serialized, Deserialized****.*
* ***Serialized*** *:The Serialized approach is mainly used to* transfer *the data through the network with each request and response.*
* ***Deserialized***

*While the deserialized approach is used to read and write data to the web token.*

*JWT in the deserialized form contains only the header and the payload. Both of them are plain JSON objects.*

*Spring boot basics \*\*\*\*\*\*\*\*\*\*\*\*\**

*The key concepts of Spring Boot include:*

* Auto-configuration: *Spring Boot automatically configures components based on dependencies and defaults.*
* Starter Dependencies*: Pre-configured dependencies that simplify dependency management for specific use cases.*
* Embedded Server: *An included server that allows running applications as standalone executables.*
* Actuator: *Provides endpoints and tools for monitoring and managing Spring Boot applications.*
* Spring Boot CLI*: Command-line tool for quickly creating and prototyping Spring Boot applications.*
* Configuration Properties*: Convenient way to configure applications using properties or YAML files.*
* Spring Boot Starters*: Simplify integration of third-party libraries or frameworks into Spring Boot applications.*
* Production-Ready Features*: Built-in features like metrics, health checks, logging, and security for production environments.*
* Spring Boot Actuator*: Provides endpoints and tools for monitoring and managing Spring Boot applications.*

## *Actuator:*

## *What :*

*provides production-ready features to monitor and manage applications.*

*It includes endpoints for health checks, metrics, logging, environment information, and more. Actuator enables better visibility and manageability of Spring Boot applications in production.*

*Spring Boot Annotations*

*: Spring Boot provides various annotations that simplify application development. These include @SpringBootApplication, which combines @Configuration, @EnableAutoConfiguration, and @ComponentScan; @RestController for creating RESTful endpoints; @Autowired for dependency injection; and more.*

*Externalized Configuration:*

*Spring Boot supports externalized configuration, allowing developers to configure applications using property files (application.properties or application.yml), environment variables, command-line arguments, or cloud configuration services. This promotes flexibility and portability.*

*Spring Boot Devtools: Spring Boot Devtools provides developer-focused tools, such as automatic restarts, live reload of static resources, and enhanced logging. It improves productivity by reducing development turnaround time.*

# Rest and sope API

* SOAP: Simple Object Access Protocol
* REST: Representational State Transfer
* *SOAP is a protocol, while REST is an architectural style.*
* *SOAP messages are typically larger and more complex, while REST messages are simpler and more lightweight.*
* *REST is stateless, whereas SOAP can have stateful interactions.*
* *REST APIs have a uniform interface using standard HTTP methods, while SOAP APIs have a standardized structure defined by the SOAP specification.*
* *REST APIs are generally easier to use and understand, while SOAP APIs can be more complex.*
* *REST APIs are often more scalable, while SOAP APIs may require more resources.*
* *REST has widespread adoption for public APIs and web applications, while SOAP is commonly used in enterprise and legacy systems.*

# *ACID*

*Transaction protocols typically try to provide ACID guarantees, where ACID is an acronym for:*

* ***A****tomicity – All changes occur or none of the changes occur – prevents partial updates.*
* ***C****onsistency – The system as a whole move from one consistent state to another consistent state.*
* ***I****solation – Changes by one transaction aren’t seen by any other transaction until the transaction is complete - sometimes referred to as serializability meaning the results are the same whether transactions execute in parallel or are serialized.*
* ***D****urability – Once the outcome of the transaction has been determined, the outcome is durably recorded and will take place even in the presence of temporary failures.*

Microservices architecture is typically suitable in the following scenarios:

Complex and large-scale applications: Microservices are beneficial when dealing with complex applications that have multiple functionalities and require scalability. Breaking down the system into smaller, independent services makes it easier to manage and develop.

Independent deployment and scalability: If different parts of your application require frequent updates, scaling, or deployment independently, microservices can be advantageous. Each microservice can be developed, tested, deployed, and scaled independently, allowing for greater flexibility.

Polyglot development: Microservices architecture enables the use of different programming languages, frameworks, and technologies for each service. This is beneficial when different teams prefer different technologies or when specific services require specialized tools or libraries.

Team autonomy and agility: Microservices enable small, autonomous teams to develop and deploy their services independently. This promotes agility, as teams can work in parallel and release updates without coordination with other teams, reducing bottlenecks and enabling faster development cycles.

Scalability and fault isolation: Microservices facilitate horizontal scaling, allowing you to scale individual services based on demand. Additionally, if one microservice fails, it won't necessarily impact the entire system, as failures are isolated to specific services.

Rapid innovation and experimentation: Microservices architecture supports rapid innovation and experimentation by enabling the development of new features or services without disrupting the entire application. This allows for faster iterations and the ability to test and validate new ideas.

## What is the role of monitoring and logging in a microservice environment?

*In a microservice environment, monitoring and logging have the following key roles:*

* + *Monitoring: Collecting and analyzing metrics to ensure system health, performance, and availability.*
  + *Logging: Centralizing logs for troubleshooting, debugging, and auditing.*
  + *Distributed Tracing: Providing visibility into requests across multiple microservices for performance optimization.*
  + *Anomaly Detection and Alerting: Identifying abnormal behavior and triggering alerts for timely investigation.*
  + *Capacity Planning and Optimization: Forecasting resource requirements and optimizing capacity.*
  + *Security Monitoring: Detecting and responding to security events or suspicious activities*

## How do you ensure data security and privacy in microservices?

* *Use secure communication protocols like HTTPS/TLS.*
* *Implement authentication and authorization mechanisms.*
* *Employ Role-Based Access Control (RBAC).*
* *Utilize data encryption for storage and transmission.*
* *Validate and sanitize incoming data.*
* *Secure configuration management of sensitive information.*
* *Implement comprehensive logging and auditing.*
* *Mask or anonymize unnecessary sensitive data.*
* *Secure service-to-service communication.*
* *Conduct regular security testing and vulnerability assessments.*
* *Ensure compliance with data protection regulations.*

## How to communicate micrservicess?

* *HTTP/REST APIs for communication.*
* *Messaging/event-driven approach using message brokers.*
* *RPC frameworks for remote procedure calls.*
* *Service mesh frameworks for seamless communication and management.*
* *API gateways as a single entry point for routing requests.*
* *Shared libraries or SDKs to encapsulate communication logic.*

## *DevTools*

*Spring Boot DevTools is a module that enhances the developer experience during application development. It provides features like automatic restart, live reload, enhanced error page, database console, and additional development-time features. DevTools improves productivity, reduces development time, and simplifies common development tasks. It is meant for use in the development phase and is not recommended for production environments.*

## *What is a Spring Boot Actuator?*

*Spring Boot Actuator is a powerful feature of the Spring Boot framework that provides a set of production-ready tools and endpoints to monitor and manage Spring Boot applications. It enables developers and system administrators to gain insights into the application's health, metrics, environment, and more, allowing for efficient monitoring and troubleshooting in a production environment.*

*Key points to mention in an interview answer about Spring Boot Actuator:*

* *Purpose: The main purpose of Spring Boot Actuator is to provide operational and management features for Spring Boot applications in production environments.*
* *Key Features: Actuator offers a range of features, including health checks, metrics, environment information, auditing and tracing, management endpoints, security and access control.*
* *Health Checks: Actuator provides built-in health indicators to monitor the application's health status. Custom health checks can be added to assess specific components or dependencies.*
* *Metrics and Monitoring: Actuator exposes various metrics about the application, such as request rates, CPU usage, memory usage, etc. It integrates with popular monitoring systems for collecting and visualizing application metrics.*
* *Environment Information: Actuator provides detailed information about the application's environment, including configuration properties, system properties, and environment variables.*
* *Auditing and Tracing: Actuator includes endpoints for auditing and tracing requests, allowing tracking and monitoring of request details for analysis and debugging.*
* *Management Endpoints: Actuator offers a set of management endpoints that expose operational information and allow interaction with the application. It enables tasks like shutting down, restarting, or viewing thread dumps.*
* *Security and Access Control: Actuator supports securing management endpoints using Spring Security, allowing access control based on roles and permissions.*
* *Customization: Actuator provides extensibility and customization options, allowing developers to add custom health indicators, metrics, or management endpoints specific to their application's requirements.*
* *Integration: Actuator seamlessly integrates with other Spring Boot features and libraries, making it easy to incorporate into existing Spring Boot projects*

## *How can you access a value defined in the application? What is a properties file in Spring Boot?*

*Use the @Value annotation to access the properties which is defined in the application – properties file.*

*@Value("${custom.value}")*

*private String customVal;*

## *What is the primary difference between Spring and Spring Boot?*

* *Spring is a web application development framework based on Java.*
* *On the other hand Spring Boot is an extension of the spring framework which eliminated the boilerplate configuration required for setup a Spring application.*

## *Explain Spring Boot Admin*

* *Spring Boot admin is a community project which helps you to manage and monitor your Spring Boot applications.*

## *Can you disable the default web server in the Spring Boot application?*

*Yes, we can disable the default web server by using application.properties to configure the web application type.*

*spring.main.web-application-type=none*

## *What is @pathVariable?*

*@PathVariable annotation helps you to extract information from the URI directly.*

## *What is filter in spring boot*

*In a Spring Boot application, filters are components that intercept and process incoming requests and outgoing responses in a web application. Filters are part of the Servlet API and allow developers to perform pre-processing and post-processing tasks on HTTP requests and responses.*

*Key points to mention in an interview answer about filters in Spring Boot:*

* *Purpose: Filters in Spring Boot are used to perform tasks such as logging, authentication, authorization, request/response modification, and more.*
* *Implementation: Filters can be implemented by implementing the javax.servlet.Filter interface or extending the javax.servlet.GenericFilterBean class.*
* *Filter Configuration: Filters can be configured in different ways, such as using annotations (@WebFilter), FilterRegistrationBean in a configuration class, or configuring in application.properties or application.yml using spring.servlet.filter.registration properties.*
* *Order: Filters can have an order defined using the setOrder() method or the order property. The order determines the sequence in which filters are applied when multiple filters are present.*
* *URL Patterns: Filters can be mapped to specific URL patterns using addUrlPatterns() or url-patterns property. The filter will only be applied to requests matching the specified patterns.*
* *Pre-processing and Post-processing: Filters provide a doFilter() method where developers can implement pre-processing logic before passing the request to the next filter or servlet, and post-processing logic after the request has been processed.*
* *Use Cases: Filters are commonly used for tasks such as request logging, authentication and authorization, request/response modification (e.g., adding headers, modifying content), and handling cross-cutting concerns.*
* *Flexibility: Filters offer a flexible mechanism to intercept and manipulate requests and responses. They can be combined with other Spring Boot features, such as security filters, to implement complex application behavior.*
* *Customization: Developers can create custom filters by implementing the Filter interface or extending the GenericFilterBean class. This allows for implementing specific business logic or integrating with external systems.*

## *What are the major differences between RequestMapping and GetMapping?*

*RequestMapping can be used with GET, POST, PUT, and many other request methods using the method attribute on the annotation. Whereas GetMapping is only an extension of RequestMapping, which helps you to improve clarity on requests.*

## *What do you mean by hot-swapping in Spring Boot?*

*It is a way to reload the changes without restarting Tomcat, or Jetty server. Eclipse and Many other IDEs support bytecode hot swapping. If you make any changes that don’t affect the method signature, it should reload without side effect.*

## *What is @crossOrigin*

*@CrossOrigin is an annotation in Spring Boot that enables Cross-Origin Resource Sharing (CORS) for specific controller methods or the entire controller class. It allows web browsers to make cross-origin requests, meaning requests from one domain to another. By using @CrossOrigin, you can specify which origins are allowed to access your controller methods, thereby controlling the CORS behavior for your Spring Boot application.*

## *Explain caching.*

*Caching is a memory are that temporary stores frequently accessed data that is otherwise expensive to get or compute.*

## *What is mean by spring batch?*

*Spring Boot Batch provides code reusability which is important when working with large numbers of records, including transaction management, logging, skipping, job processing statistics, and job restarts.*

## *Explain Apache Kafka.*

*Apache Kafka is an open-source messaging platform. LinkedIn develops it. Apache Kafka enables the user to build distributed applications and handle real-time data feeds. Kafka is suitable for both offline and online messaging*

## *Explain different types of dependency injection.*

## *There are two types of dependency injection in Spring Boot. They are as follows:*

## ***Constructor-based dependency injection:****It is a technique in which one class object supplies the dependency of another object.*

## ***Setter-based dependency injection:****It is a dependency injection in which the framework injects the primitive and string-based values using setter method.*

## *Internal working of set*

*The internal working of the Set interface in Java depends on the specific implementation being used. The Set interface is part of the Java Collections Framework and provides a collection of unique elements with no defined order.*

*Here are some common implementations of the Set interface and their internal workings:*

*HashSet: HashSet is a widely used implementation of the Set interface. It internally uses a hash table to store the elements. When an element is added to a HashSet, its hash code is computed, and the element is stored in a bucket corresponding to that hash code. If multiple elements have the same hash code, they are stored in the same bucket as a linked list. HashSet provides constant-time performance for the basic operations like adding, removing, and checking for the presence of an element.*

*TreeSet: TreeSet is an implementation of the Set interface that provides a sorted set of elements in ascending order. It internally uses a balanced binary search tree, typically a red-black tree, to store the elements. The elements are ordered based on their natural ordering or a custom comparator provided at the time of creation. TreeSet provides logarithmic-time performance for basic operations like adding, removing, and searching for an element.*

*LinkedHashSet: LinkedHashSet is an implementation of the Set interface that maintains the insertion order of elements. It internally uses a combination of a hash table and a linked list to store the elements. The hash table provides constant-time performance for adding, removing, and searching for an element, while the linked list maintains the order of insertion.*

*EnumSet: EnumSet is a specialized implementation of the Set interface specifically designed for use with enum types. It is highly optimized for enum values and provides constant-time performance for basic operations. Internally, EnumSet uses a bit vector to represent the set of enum values.*

## *paging in java in short*

## What is json in java

*In Java, JSON (JavaScript Object Notation) is a popular format for exchanging and representing structured data. It provides libraries and APIs for parsing, generating, and manipulating JSON data . With JSON in Java, you can easily convert JSON to Java objects and vice versa, validate JSON against schemas, and integrate with JSON-based web services. It simplifies data interchange and enables seamless interaction with JSON-based systems.*

*JWT java*

*Steps :*

*Implement websecurityconfigadapter*

*override*

*🡪Configuration ()*

*🡪Configure()*

*Implement userdetialservise()*

*🡪userdetial();*

*🡪*

# *Java interview question*

In Spring, the **scope** of a bean determines the lifecycle and visibility of the bean within the Spring container. Bean scopes define how Spring manages the instantiation, sharing, and lifecycle of beans. Spring provides several types of bean scopes to cater to different use cases.

### **Bean Scopes in Spring**

1. **Singleton** (@Scope("singleton"))
   * **Description**: This is the default scope. A single instance of the bean is created and shared across the entire Spring container. All requests for this bean get the same instance.
   * **Usage**: Suitable for stateless beans where a single shared instance is sufficient and efficient.
   * **Lifecycle**: Created once per Spring container lifecycle, initialized at application startup.
2. **Prototype** (@Scope("prototype"))
   * **Description**: A new instance of the bean is created each time it is requested from the container. The container does not manage the bean's lifecycle beyond its creation.
   * **Usage**: Ideal for beans where a new instance is required for each use, such as stateful beans or beans that require unique instances.
   * **Lifecycle**: Created every time it is requested, and not managed beyond creation (e.g., no destruction callback).
3. **Request** (@Scope("request"))
   * **Description**: A new instance of the bean is created for each HTTP request. The bean is scoped to the lifecycle of a single HTTP request.
   * **Usage**: Commonly used in web applications where you need a new instance per HTTP request, such as user sessions or request-specific data.
   * **Lifecycle**: Created and destroyed with each HTTP request.
4. **Session** (@Scope("session"))
   * **Description**: A new instance of the bean is created for each HTTP session. The bean is scoped to the lifecycle of an HTTP session.
   * **Usage**: Useful in web applications where you need to maintain state or user-specific data across multiple requests within a single session.
   * **Lifecycle**: Created and destroyed with each HTTP session.
5. **Application** (@Scope("application"))
   * **Description**: A single instance of the bean is created and shared across the entire ServletContext. It is similar to the Singleton scope but specific to web applications.
   * **Usage**: Used when you need a bean to be shared across all users and sessions in a web application.
   * **Lifecycle**: Created once per ServletContext lifecycle, similar to a singleton but within the web application context.
6. **WebSocket** (@Scope("websocket"))
   * **Description**: A new instance of the bean is created for each WebSocket session. The bean is scoped to the lifecycle of a WebSocket session.
   * **Usage**: Suitable for beans that handle WebSocket connections and require instance-specific data.
   * **Lifecycle**: Created and destroyed with each WebSocket session.

### **Summary**

* **Singleton**: One instance per Spring container (default).
* **Prototype**: New instance for each request.
* **Request**: New instance for each HTTP request (web applications only).
* **Session**: New instance for each HTTP session (web applications only).
* **Application**: One instance per ServletContext (web applications only).
* **WebSocket**: New instance for each WebSocket session (web applications only).

JAVA DESIGN PATTERN

Design patterns are standard solutions to common problems in software design. They provide reusable solutions and best practices to address recurring design challenges. In Java, design patterns are categorized into three main types: **Creational**, **Structural**, and **Behavioral**. Here’s an overview of some common design patterns in each category:

### **Creational Patterns**

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

1. **Singleton**
   * **Purpose**: Ensures that a class has only one instance and provides a global point of access to it.
   * **Usage**: Used when exactly one instance of a class is needed to coordinate actions across the system.
2. **Factory Method**
   * **Purpose**: Defines an interface for creating an object, but allows subclasses to alter the type of objects that will be created.
   * **Usage**: Useful when a class can’t anticipate the class of objects it must create.
3. **Abstract Factory**
   * **Purpose**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
   * **Usage**: Used when the system needs to be independent of how its products are created, composed, and represented.
4. **Builder**
   * **Purpose**: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
   * **Usage**: Useful for creating complex objects with many optional components.
5. **Prototype**
   * **Purpose**: Creates new objects by copying an existing object, known as the prototype.
   * **Usage**: Used when creating an instance of a class is more expensive than copying an existing instance.

### **Structural Patterns**

Structural patterns deal with object composition or the structure of classes and objects.

1. **Adapter (or Wrapper)**
   * **Purpose**: Allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces.
   * **Usage**: Used to integrate new systems with legacy systems that have incompatible interfaces.
2. **Decorator**
   * **Purpose**: Adds new functionality to an object dynamically without altering its structure.
   * **Usage**: Useful for extending the functionalities of classes in a flexible and reusable way.
3. **Facade**
   * **Purpose**: Provides a simplified interface to a complex subsystem.
   * **Usage**: Used to make a subsystem easier to use by providing a unified interface.
4. **Composite**
   * **Purpose**: Allows individual objects and composites of objects to be treated uniformly.
   * **Usage**: Used to represent part-whole hierarchies where clients can treat individual objects and compositions of objects uniformly.
5. **Bridge**
   * **Purpose**: Separates an abstraction from its implementation so that the two can vary independently.
   * **Usage**: Useful when both the class and what it does vary frequently.

### **Behavioral Patterns**

Behavioral patterns deal with object interactions and responsibilities.

1. **Observer**
   * **Purpose**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   * **Usage**: Used when a change in one object requires changing others, and you want to keep them loosely coupled.
2. **Strategy**
   * **Purpose**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
   * **Usage**: Used to define a strategy for a particular task and allow the strategy to be chosen dynamically at runtime.
3. **Command**
   * **Purpose**: Encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations.
   * **Usage**: Useful for implementing undo functionality or handling requests in a queue.
4. **Chain of Responsibility**
   * **Purpose**: Passes a request along a chain of handlers, allowing multiple objects to handle the request without coupling the sender to the receiver.
   * **Usage**: Used to process a request through a chain of handlers where each handler can process or forward the request.
5. **Mediator**
   * **Purpose**: Defines an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from referring to each other explicitly.
   * **Usage**: Used to centralize complex communication and control logic between multiple objects.
6. **Template Method**
   * **Purpose**: Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
   * **Usage**: Used to allow subclasses to redefine certain steps of an algorithm without changing the algorithm’s structure.
7. **State**
   * **Purpose**: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
   * **Usage**: Used when an object should change its behavior based on its state.
8. **Iterator**
   * **Purpose**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
   * **Usage**: Used to iterate over collections of objects without exposing their internal structure.
9. **Memento**
   * **Purpose**: Captures and externalizes an object’s internal state without violating encapsulation, allowing the object to be restored to this state later.
   * **Usage**: Used to implement undo functionality and restore previous states.
10. **Visitor**
    * **Purpose**: Defines a new operation to a group of objects without changing the classes of the objects.
    * **Usage**: Useful for operations that need to be applied to a collection of objects with different types.

Each of these patterns solves a particular design problem and can be used to improve code maintainability, flexibility, and reusability in Java applications.

Top of Form

What is circuit breaker in microservice   
In microservices architecture, a circuit breaker is a design pattern that prevents system failure when a service stops working properly. It works by monitoring calls to a service and stopping further calls if too many fail. This gives the system time to recover without being overwhelmed with requests. Once the service is stable, the circuit breaker resets and calls can go through again.

Bottom of Form

What is saga pattern

The **Saga Design Pattern** is a pattern used to manage long-running transactions and ensure data consistency across microservices. It is especially useful in distributed systems where a single transaction spans multiple services, which can be challenging to manage due to their independent nature.

### **Key Concepts of the Saga Pattern**

1. **Long-Running Transactions**:
   * In a microservices architecture, a business process often involves multiple services, each managing its own data. A saga helps manage transactions that span multiple services.
2. **Series of Steps**:
   * A saga consists of a sequence of local transactions, each performed by a different microservice. Each local transaction must be completed successfully before moving to the next one.
3. **Compensating Transactions**:
   * If a step in the saga fails, compensating transactions are executed to undo the changes made by previous steps. This ensures that the system remains in a consistent state even if part of the saga fails.
4. **Two Main Types**:
   * **Choreography**: Each service involved in the saga knows about the other services and directs the saga flow. Services emit events to notify other services of their state and actions.
   * **Orchestration**: A central coordinator service directs the saga. It sends commands to each service, coordinates the sequence of transactions, and handles failure and compensation.

### **Benefits of the Saga Pattern**

* **Data Consistency**: Ensures that all services involved in a transaction maintain data consistency even in case of partial failures.
* **Resilience**: Improves the system's resilience by providing mechanisms to handle failures and rollbacks.
* **Decoupling**: Allows microservices to remain loosely coupled, with each service managing its own local transactions and compensations.

### **Example Use Case**

Consider an e-commerce system where placing an order involves:

1. **Inventory Service**: Reduces stock.
2. **Payment Service**: Processes payment.
3. **Shipping Service**: Ships the product.

In a saga:

* **Step 1**: Inventory Service reduces stock.
* **Step 2**: Payment Service processes payment.
* **Step 3**: Shipping Service ships the product.

If Payment Service fails after the Inventory Service has already reduced stock, a compensating transaction will be triggered to restore the stock level, ensuring overall consistency.

The Saga pattern effectively manages distributed transactions, making it a crucial design pattern for building reliable and resilient microservices architectures.

Top of Form

Bottom of Form