**What is an IOC container?**

An **IoC (Inversion of Control)** Container in Spring Boot is essentially a central manager for the application objects that controls the creation, configuration, and management of dependency injection of objects (often referred to as beans), also referred to as a DI (Dependency Injection) container.

IoC Container is a framework for implementing automatic dependency injection. It manages object creation and its life-time and also injects dependencies into the class.

**. What is dependency Injection and its types?**

**Dependency Injection** (DI) is a design pattern that enables us to produce loosely coupled components. In DI, an object’s ability to complete a task depends on another object. There three types of dependency Injections.

* **Constructor injection:** This is the most common type of DI in Spring Boot. In constructor injection, the dependency object is injected into the dependent object’s constructor.
* **Setter injection:** In setter injection, the dependency object is injected into the dependent object’s setter method.
* **Field injection**: In field injection, the dependency object is injected into the dependent object’s field.
* The process of injecting dependent bean objects into target bean objects is called dependency injection.
* Setter Injection: The IOC container will inject the dependent bean object into the target bean object by calling the setter method.
* Constructor Injection: The IOC container will inject the dependent bean object into the target bean object by calling the target bean constructor.
* Field Injection: The IOC container will inject the dependent bean object into the target bean object by Reflection API.

**. What are the @RequestMapping and @RestController annotations in Spring Boot used for?**

**@RequestMapping:** @RequestMapping is used to map HTTP requests to handler methods in your controller classes. It can be used at the class level and method level. It supports mapping by:

* HTTP method – GET, POST, PUT, DELETE
* URL path
* URL parameters
* Request headers
* **@RequestMapping:**This annotation is used to map HTTP requests to a specific method in a controller. It can be applied at the class level to define a base URL for all methods in the class, or at the method level to specify a specific URL mapping.

**@RestController:**@RestController is a convenience annotation that combines **@Controller** and **@ResponseBody**. It indicates a controller where every method returns a domain object instead of a view.

*@RestController = @Controller + @ResponseBody*

*OR*

**@RestController** annotation is like a shortcut to building RESTful services. It combines two annotations:

* **@Controller**: Marks the class as a request handler in the Spring MVC framework.
* **@ResponseBody**: Tells Spring to convert method return values (objects, data) directly into HTTP responses instead of rendering views.

It enables us to Define endpoints for different **HTTP methods (GET, POST, PUT, DELETE),**return data in various formats (JSON, XML, etc.) and map the request parameters to method arguments.

**. What is the difference between @RestController and @Controller in Spring Boot?**

@Controller Map of the model object to view or template and make it human readable but @RestController simply returns the object and object data is directly written in HTTP response as JSON or XML.

**@ComponentScan** annotation is used to tell Spring to scan a package and automatically detect Spring components, configurations, and services to configure.The @ComponentScan annotation can be used in the following ways:

* **Without arguments**
* **What is Spring Boot dependency management?**
* **Spring Boot dependency management** makes it easier to manage dependencies in a Spring Boot project. It makes sure that all necessary dependencies are appropriate for the current Spring Boot version and are compatible with it.

Spring Boot dependency management is used to manage dependencies and configuration automatically without you specifying the version for any of that dependencies.

.

* **@Component:** This annotation is the most generic annotation for any Spring-managed component. It is used to mark a class as a Spring bean that will be managed by the Spring container.

**What does the @SpringBootApplication annotation do internally?**

The **@SpringBootApplication** annotation combines three annotations. Those three annotations are: **@Configuration, @EnableAutoConfiguration,**and **@ComponentScan**.

* **@AutoConfiguration**: This annotation automatically configuring beans in the class path and automatically scans the dependencies according to the application need.
* **@ComponentScan**: This annotation scans the components (@Component, @Service, etc.) in the package of annotated class and its sub-packages.
* **@Configuration:** This annotation is used to indicate that a class contains configuration methods for the application context. It is typically used in combination with @Bean annotations to define beans and their dependencies.

@SpringBootApplication automatically configures the application based on the dependencies added during project creation and bootstraps the application by using run() method inside the main class of an application.

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

* **What is Spring Initializr?**
* **Spring Initializer** is a tool that helps us to create skeleton of spring boot project or project structure by providing a maven or gradle file to build the application. It set up the framework from scratch.

**. Explain the internal working of Spring Boot.**

**Here are the main steps involved in how Spring Boot works:**

* Start by creating a new Spring Boot project.
* Add the necessary dependencies to your project.
* Annotate the application with the appropriate annotations.
* Run the application.

**. How does a spring application get started?**

A Spring application gets started by calling the **main()** method with **@SpringBootApplication**annotation in the **SpringApplication** class. This method takes a SpringApplicationBuilder object as a parameter, which is used to configure the application.

* Once the SpringApplication object is created, the **run()** method is called.
* Once the application context is initialized, the run() method starts the application’s embedded web server.

**Example:**

Java

**import** **org.springframework.boot.SpringApplication**;

**import** **org.springframework.boot.autoconfigure.SpringBootApplication**;

@SpringBootApplication

**public** **class** **MyApplication**

{

**public** **static** void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

**. Describe the flow of HTTPS requests through the Spring Boot application.**

The flow of HTTPS requests through a Spring Boot application is as follows:

* First client makes an **HTTP request**(**GET, POST, PUT, DELETE**) to the browser.
* After that the request will go to the controller, where all the requests will be mapped and handled.
* After this in Service layer, all the **business logic** will be performed. It performs the business logic on the data that is mapped to **JPA (Java Persistence API)**using model classes.
* In repository layer, all the **CRUD** operations are being done for the **REST APIs**.
* A **JSP page** is returned to the end users if no errors are there.

**What is Spring Bean?**

An object that is managed by the Spring IoC container is referred to as a spring bean. A Spring bean can be any Java object.

**Dependency Injection in Spring Boot**

Dependency injection (DI) is a fundamental concept in Spring Boot, facilitating loose coupling and testability. Spring Boot offers several annotations for DI, such as **@Autowired**, **@Qualifier**, and **@Value**. Key points about DI include:

* Dependency injection allows objects to be provided with their dependencies rather than creating them internally.
* Inversion of Control (IoC) is a core principle behind DI, where control of object creation and dependency management is delegated to an IoC container (Spring).
* DI promotes modularity, reusability, and testability by decoupling components.

**Spring Boot Autoconfiguration**

Autoconfiguration is a powerful feature of Spring Boot that automatically configures beans and infrastructure based on classpath dependencies and properties. Key points about autoconfiguration include:

* Spring Boot autoconfiguration uses the **@Conditional** annotations to conditionally enable or disable configurations based on the presence or absence of certain classes or properties.
* Autoconfigured beans are automatically instantiated, wired, and ready for use without explicit configuration.
* You can exclude specific autoconfigurations using the **@EnableAutoConfiguration** annotation's **exclude** attribute.

**Answer:**Auto-configuration is one of the most powerful features of Spring Boot. It automatically configures your Spring application based on the jar dependencies that you have added. For instance, if your project includes spring-boot-starter-web, Spring Boot automatically configures your application to use Tomcat as the default web server and configures Spring MVC for web development. This feature is implemented by examining your classpath and the beans you have defined, making intelligent decisions on what configuration is likely required. Developers can override the default settings by specifying their configurations.

**Spring Boot Starters**

Spring Boot starters are dependency descriptors that simplify the inclusion of common dependencies for various functionalities. They provide a convenient way to manage dependencies and ensure compatibility. Some popular starters include:

* **spring-boot-starter-web**: Includes dependencies for building web applications.
* **spring-boot-starter-data-jpa**: Provides dependencies for working with JPA-based data access.
* **spring-boot-starter-test**: Contains dependencies for testing Spring Boot applications.

By utilizing Spring Boot annotations and concepts effectively, you can enhance the development and configuration process.

-**Answer:**Spring Boot starters are a set of dependency descriptors that simplify the Maven or Gradle dependencies you need to add to your project. They aggregate common dependencies into single, version-managed artifacts. This way, developers can avoid specifying each dependency individually and ensure version compatibility across the Spring ecosystem. For example, spring-boot-starter-web is used for building web applications, including RESTful applications using Spring MVC. It includes Tomcat as the default embedded container, Spring MVC, and other necessary libraries. Another example is spring-boot-starter-data-jpa for Spring Data JPA with Hibernate, which simplifies the data access layer configuration and integration.

--------------------------

‍

**JPA (Java Persistence API) and Spring Data JPA**

Java Persistence API (JPA) is a standard specification for ORM (Object-Relational Mapping) in Java applications. Spring Data JPA, built on top of JPA, provides additional abstractions and simplifies the implementation of JPA-based data access. Key points to understand:

* JPA allows mapping Java objects to database tables and provides a convenient way to perform CRUD (Create, Read, Update, Delete) operations.
* Spring Data JPA reduces boilerplate code by providing repository interfaces with pre-defined methods for common database operations.
* By utilizing JPA and Spring Data JPA, you can easily work with databases in your Spring Boot applications.

**Exception Handling in Spring Boot**

Error handling is an essential aspect of RESTful APIs. Spring Boot simplifies exception handling through the use of global and controller-specific exception handlers. Key points include:

* **@ControllerAdvice**: Annotate a class with **@ControllerAdvice** to define global exception handlers.
* **@ExceptionHandler**: Annotate methods with **@ExceptionHandler** to handle specific exceptions.
* Return appropriate error responses using the **ResponseEntity** class or custom error DTOs.

## Spring Boot Actuator

Spring Boot Actuator is a powerful feature that provides monitoring, management, and operational endpoints for Spring Boot applications. Let's explore its capabilities and how to leverage them effectively.

---------------------------------------------------------------------

"In Spring Boot, we can handle exceptions in RESTful APIs by using the **@ExceptionHandler** annotation. By annotating methods with **@ExceptionHandler**, we can handle specific exceptions and return appropriate error responses to clients. Additionally, we can use **@ControllerAdvice** to define global exception handlers that apply across multiple controllers. It's essential to return meaningful error responses, including appropriate HTTP status codes and error messages, to help clients understand the encountered issue."

### Monitoring and Managing Applications with Actuator

Actuator enables real-time monitoring and management of Spring Boot applications by exposing various endpoints. Key features include:

* **Health Checks**: The **/health** endpoint provides information about the application's health, including database connectivity, disk space, and custom health indicators.
* **Metrics**: Actuator exposes metrics about the application's performance, such as request count, response time, memory usage, and more.
* **Info Endpoint**: The **/info** endpoint provides custom application-specific information.
* **Auditing**: Actuator supports auditing features, allowing you to track and monitor important events within your application.

### Built-in Endpoints and Metrics

Spring Boot Actuator comes with several built-in endpoints and metrics that provide valuable insights into your application's health and performance. Some notable built-in endpoints include:

* **/health**: Provides health-related information, including the status of various components.
* **/info**: Displays custom application-specific information.
* **/metrics**: Exposes various metrics about your application, such as memory usage, CPU utilization, and request statistics.

### Customizing Actuator Endpoints

Spring Boot Actuator allows you to customize and secure the exposed endpoints according to your application's requirements.

* **Endpoint Configuration**: Use the **management.endpoints.web.exposure.include** or **management.endpoints.web.exposure.exclude** property to configure which endpoints to include or exclude.
* **Endpoint Security**: Secure Actuator endpoints by configuring authentication and authorization mechanisms.
* **Custom Endpoints**: Create custom endpoints by implementing the **Endpoint** interface or extending the **AbstractEndpoint** class.

Spring Boot Actuator empowers you to monitor and manage your applications effectively.

‍"Spring Boot Actuator is a feature of Spring Boot that provides monitoring and management capabilities for applications. It exposes endpoints that allow real-time monitoring and management of different aspects of the application. For example, the **/health** endpoint provides health checks to determine the application's overall health, and the **/metrics** endpoint collects and exposes various application metrics. Actuator helps in real-time monitoring, health checks, and performance measurement, providing valuable insights into the application's status and behavior."

### Introduction to Spring Security

Spring Security is a powerful framework that provides authentication, authorization, and other security features for Java applications. Spring Boot seamlessly integrates with Spring Security, making it easy to secure your applications.

### Securing a Spring Boot Application

To secure your Spring Boot application, you need to configure authentication and authorization mechanisms. Here's a step-by-step approach:

1. **Dependency Inclusion**: Include the **spring-boot-starter-security** dependency in your project.
2. **Default Security Configuration**: Spring Boot automatically configures basic security features, including a default login page and in-memory user details.
3. **User Authentication**: Customize user authentication by configuring user details, password encoding, and authentication providers.
4. **Authorization**: Define authorization rules using **antMatchers** and **hasAuthority** to restrict access to specific resources or URLs.
5. **Securing RESTful APIs**: Apply security to RESTful APIs using authentication filters, tokens, or OAuth2.

### Authentication and Authorization in Spring Boot

Authentication and authorization are essential aspects of securing applications. In Spring Boot, you can achieve these through various mechanisms:

* **Authentication**: Spring Security supports authentication mechanisms like form-based login, HTTP Basic authentication, and token-based authentication.
* **Authorization**: Spring Security provides annotations like **@PreAuthorize** and **@Secured** for method-level authorization, and **antMatchers** for URL-level authorization.

### 13. How do you secure a Spring Boot application?

"To secure a Spring Boot application, we can leverage the features of Spring Security, which is a powerful security framework for Java applications. We include the **spring-boot-starter-security** dependency in our project, which provides the necessary components and configurations for securing the application. We define authentication mechanisms, such as form-based login or token-based authentication, and configure authorization rules to control access to different resources. Additionally, we can secure RESTful APIs by implementing authentication filters or utilizing OAuth2 for token-based authentication."

### 1. What is Spring Boot, and why is it popular?

"Spring Boot is a framework that provides a streamlined way to create production-ready Spring applications with minimal configuration. It offers features like auto-configuration, which reduces the need for manual setup, and embedded servers, eliminating the need for deploying applications to separate servers. Spring Boot also simplifies dependency management, making it easier to manage and resolve dependencies. Its popularity stems from its ability to boost developer productivity and accelerate the development process."

### 2. What are Spring Boot starters?

"Spring Boot starters are dependency descriptors that simplify including commonly used dependencies in a Spring Boot project. They provide a convenient way to manage dependencies and ensure that the required libraries for specific functionalities are included in the project. By including a starter, developers don't need to manually add individual dependencies, reducing the effort required to set up a project."

### ‍3. How do you externalize configuration in Spring Boot?

"In Spring Boot, we can externalize configuration by using property files like application.properties or application.yml. These files allow us to store configuration properties separately from the codebase, making it easier to modify configurations without changing the application's source code. Additionally, Spring Boot allows us to override these configurations using environment variables or command-line arguments, providing flexibility in different deployment scenarios."

### 4. How do you define profiles in Spring Boot?

"In Spring Boot, profiles allow us to customize the application's behavior for different environments, such as development, testing, or production. We can define profiles using the **@Profile** annotation on specific beans or components, or by setting the **spring.profiles.active** property in the configuration files. By leveraging profiles, we can apply profile-specific configurations to adapt the application's behavior based on the active profile."

### 6. How do you use JPA in Spring Boot?

"In Spring Boot, we can use JPA (Java Persistence API) for object-relational mapping. JPA provides a standard way to map Java objects to relational database tables. Spring Boot simplifies the usage of JPA by providing auto-configuration and support for various JPA providers, such as Hibernate. Additionally, Spring Boot integrates seamlessly with Spring Data JPA, which extends JPA and provides additional abstractions and functionality to simplify data access."

**Key Features and Benefits of Spring Boot**

Spring Boot, built on top of the Spring Framework, is designed to make it easy to create stand-alone, production-grade Spring-based applications. It offers several key features and benefits, including:

* **Autoconfiguration**: Spring Boot provides automatic configuration based on sensible defaults, reducing the need for manual configuration.
* **Embedded Servers**: Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, eliminating the need for deploying applications to separate servers.
* **Dependency Management**: Spring Boot manages the dependencies required for your application, ensuring consistent and compatible versions.
* **Production-Ready Monitoring**: Spring Boot Actuator provides production-ready features like health checks, metrics, and monitoring endpoints.
* **Easy Deployment**: With Spring Boot's self-contained nature, deploying applications becomes effortless.