SPRING BOOT INTERVIEW QUESTIONS:

1.What is Spring Boot, and how does it differ from the Spring framework?

Spring Boot is a framework that simplifies the development of Spring-based applications.

It provides a number of features that make it easier to create stand-alone, production-grade applications, such as:

Auto-configuration: Spring Boot automatically configures many of the Spring features that you would typically need to configure manually.

Starter dependencies: Spring Boot provides starter dependencies that make it easy to add common features to your application, such as web development, data access, and messaging.

Embedded servers: Spring Boot provides embedded servers, such as Tomcat and Jetty, so you don't need to configure a separate web server.

Metrics and health checks: Spring Boot provides metrics and health checks that you can use to monitor your application.

Spring boot is built on the top of the Spring framework, but it takes a different approach to development. Spring Boot is designed to be “opinionated” which means that it makes some assumptions about how you want to develop your application. This can make it easier to get started, but it can also limit your flexibility.

Feature	Spring Boot	Spring Framework
Opinionated	yes	No
Auto-configuration	yes	No
Starter dependencies	yes	No
Embedded servers	yes	No
Metrics health checks	yes	No
flexibility	less	More

Configuration: Spring Boot focuses on convention-over-configuration, providing sensible default configurations for various components. It minimizes the need for explicit configuration files and reduces the overall configuration effort. The Spring framework, on the other hand, requires more explicit configuration through XML or Java-Based configuration files.

In general, Spring boot is a good choice for developers who want to get started with Spring quickly and easily. If you need more flexibility, then you may want to use the spring framework directly. It reduces configuration overhead and improves developer productivity.

2.How do you configure a database connection in Spring Boot?

There are two ways to configure a database connection in Spring Boot:

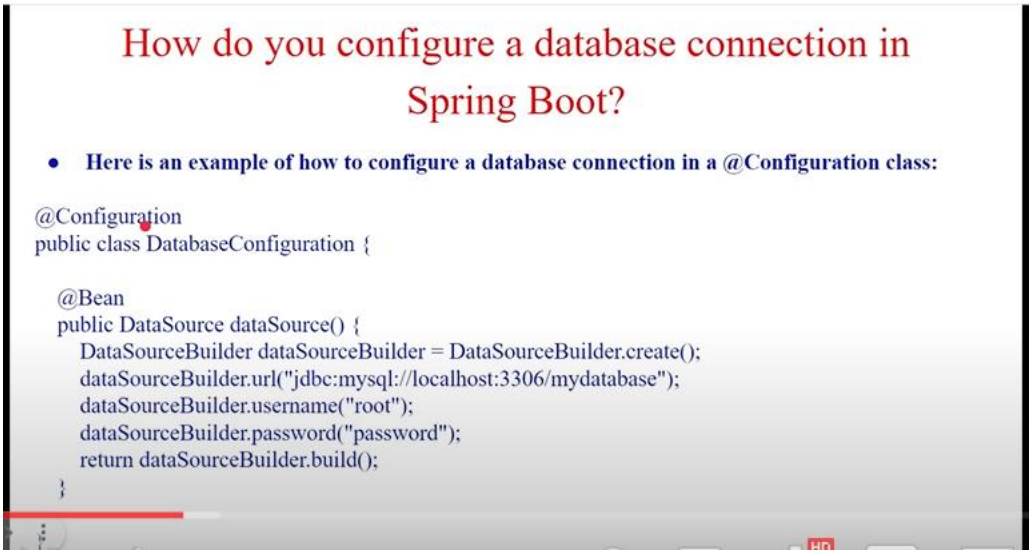
Using the application.properties file: This is the most common way to configure a database connection. You can specify the connection properties in the application.properties file, and Spring Boot will automatically configure the database connection.

In application.properties file
Example 1: To change your port number
server.port=8989
Example 2: To define the name of our application
To define the name of our application you can write the properties like this
spring.application.name = userservice
So you can see this represents the property as key-value pair here, every key associated with a value also.
Example 3: Connecting with the MySQL Database
To connect with the MySQL Database you have to write a bunch of lines. You can write the properties like this
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
spring.datasource.driver-class-name =com.mysql.jdbc.Driver

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/restaurantdb
    password: root
    username: root
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
```

Using the @Configuration class: This is a more advanced way to configure a database connection. You can create a @Configuration class and specify the connection properties in the class. Spring Boot will then use the @Configuration class to configure the database connection.



Once you have configured the database connection, you can use it in your Spring Boot application using Hibernate / JPA or even simpler JdbcTemplate.

3.How do you configure a database connection in Spring Boot?

Here is an example of how to configure a database connection in the application.properties file: spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

spring.datasource.username=root spring.datasource.password=password

4. What is the purpose of the @SpringBootApplication annotation in a Spring Boot application?

The @SpringBootApplication annotation is a convenience annotation that combines three annotations in a Spring Boot application:

@EnableAutoConfiguration: This enables Spring Boot's autoconfiguration mechanism.

Auto-configuration refers to creating beans automatically by scanning the classpath.

@ComponentScan: This annotation tells Spring Boot to scan the current package and its sub-packages in order to identify annotated classes and configure them as Spring beans.

@Configuration: This annotation designates the class as a configuration class for Java configuration.

Notes: @SpringBootApplication is annotated on the root class, inside the root package and all other classes and packages are under the root class, and they are termed as sub packages. @ComponentScan says I am gonna scan all the base package and all the sub packages, As you scan the packages and creates a bean for a spring container.

- In other words, the @SpringBootApplication annotation tells the Spring Boot to:
- Automatically configure the application based on the dependencies that are present on the classpath.
- Scan the classpath for annotated classes and configure them as Spring beans.
- Allow for custom configuration to be provided in the same class.
- This annotation makes it very easy to create a Spring Boot application, as you don't need to explicitly configure all of the beans that are needed. This can save a lot of time and effort and it can also make your application more portable.
- It will create a Spring Boot application that automatically configures itself and scans the classpath for annotated classes. When you run this application, it will start up and be ready to use.

4.How do you implement caching in Spring Boot?

- To implement caching in Spring Boot, you can follow these steps:
- Add the necessary dependencies: Include the Spring Boot starter cache dependency in your projects pom.xml file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
  <version>3.1.2</version>
</dependency>
```

- Enable caching in your Spring Boot application: Add the **@EnableCaching** annotation to your application's main class or anu configuration class:
  @SpringBootApplication
  @EnableCaching
  Public class Yourapplication{//…}

Configure a cache manager: Spring Boot provides a default cache manager, but you can also define a custom cache manager. To use the default cache manager, no further configuration is required. However, if you want to define a custom cache manager, create a bean of type CacheManager in your configuration class:

```
@Configuration
public class CacheConfig {
   @Bean
    public CacheManager cacheManager() {
    // Define and configure your cache manager here
} }
```

Add caching annotations to your methods: To enable caching for a specific method, use annotations such **as @Cacheable, @CachePut, or @CacheEvict** on the method.
For example:

```
@Service
public class YourService {
 @Cacheable("yourCacheName")
  public String getCachedData(String key) {
   // This method will be cached based on the cache name and the key parameter
   // Return the data to be cached
} }
```

In the above example, the @Cacheable annotation ensures that the getCachedData method is cached. Subsequent invocations with the same cache name and key will retrieve the cached result instead of executing the method again. Additionally, you can configure cache-specific settings such as eviction policies, time-to-live, and cache names.

5.Can you explain the different types of caching mechanisms available?
There are several types of caching mechanisms available in general, and Spring Boot supports some of the commonly used ones. Here are the main types of caching mechanisms:

In-Memory Caching: In this mechanism, the **cache data is stored in memory**, allowing for fast access and retrieval. **In Spring Boot, the default cache implementation is an in-memory cache using a ConcurrentHashMap**. This type of caching is suitable for applications where cached data is not expected to change frequently and can fit within the available memory.

Distributed Caching: Distributed caching **involves storing cache data across multiple nodes in a distributed environment.** It allows for scalling the caching ability and provides high availability. Popular distributed caching framework used with Spring Boot include **Redis** and **Hazelcast.**

External Caching: External caching involves using external caching service or platform, separate from the application. Examples of external caching services include Memcached and **Ehcache**. These services **provide dedicated caching capabilities** and can be integrated with Spring Boot applications.

Page Caching: Page caching involves caching the entire HTML pages of a website or web application. It is typically used for static content that does not change frequently. Spring Boot supports page caching through integrations with web servers and proxies like Apache HTTP Server or Nginx.

Query Caching: Query caching is used to cache the results of data queries. It can significantly **improve the performance of database-heavy applications**. Spring Boot integrates with ORM frameworks like Hibernate, which provide query capabilities out of the box.

6. What is the significance of the @Autowired annotation in Spring Boot?

- The @Autowired annotation in Spring Boot is **used for automatic dependency injection**. It is a key component of the **Inversion of Control (IoC)** principle that Spring is built upon.
- By annotating a field, constructor, or setter method with @Autowired, **Spring Boot automatically resolves and injects the appropriate dependency into the annotated component**.
- Eg There is a car class and engine class, suppose a car class requires engine class, need to inject engine class by using @autowired annotation, Both are loosely coupled , car can exist on your own and engine class can exist on your own.
- It simplifies the process of wiring dependencies between components. **Instead of manually Instantiating** or looking up dependencies, Spring Boot takes care of resolving and injecting the required dependencies at runtime.
- You can **write cleaner and more maintainable code.** It helps in achieving **loose coupling** between components, as the dependencies are managed by the Spring container rather than being hard coded in the code. This makes your code **more modular, easier to test and promotes better separation of concerns.**
- **@Autowired** supports the concept of Dependency Injection (DI), which allows for easy swapping of dependencies and promotes code reusability. It enables you to easily switch implementations or mock dependencies during unit testing or when different implementations are needed for different environments or usecases.
- Overall, the @Autowired annotation in Spring Boot plays a crucial role in simplifying dependency injection and promoting loosely coupled, modular and testable code.
-

7.How can you Handle exceptions in a Spring Boot application?

In a Spring Boot application, we can handle exceptions using various approaches. Here are some common techniques:

Global Exception handling - **@ControllerAdvice and @ExceptionHandler**:

You can create a global exception handler by using the @ControllerAdvice annotation on a class. Inside this class, you can define methods annotated with @ExceptionHandler to handle specific exceptions. These methods can provide custom logic for handling exceptions, such as returning a specific HTTP response, redirecting to an error page or logging the exception details.

Custom Exception Handling: You can create custom exception classes by extending the RuntimeException or Exception class. These custom exceptions can contain additional information specific to your application's context. You can catch these exceptions and handle them in the appropriate exception handlers.

Using try-catch exception:

8. Explain the concept of profiles in Spring Boot and how you can use them.

In Spring Boot, profiles are a **way to define different configurations for different environments** or scenarios. They allow you to customize the behavior of your application based on the specific profile that is **active at runtime**.

Profiles are helpful when you need to have different sets of configuration properties, beans or other components for various deployment environments like development, testing, production, or specific use cases.

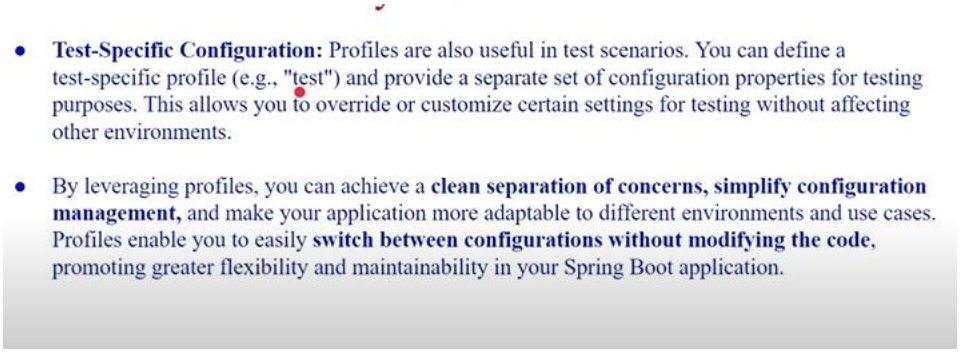
Here's how profiles work in Spring Boot:

Defining Profiles: Profiles can be defined in the application.properties or application.yml file using the spring.profiles.active property. For example, **spring.profiles.active=dev** sets the active profile to "dev". You can define multiple profiles by separating then with commas, such as spring.profiles.active=dev,qa

Profile-Specific Configuration: To create profile-specific configuration properties, you can use separate property files for each profile. For example, application-dev.properties or application-dev.yml for the "dev " profile. These files should be placed in the classpath(e.g in the src/main/resources folder). Spring Boot automatically load the properties from the active profile-specific file.

Conditional Bean Registration: You can conditionally register beans based on the active profile using the **@Profile** annotation. By annotating a bean class or method with @Profile("profile_name", **that bean will only be registered and available if the specific profile is active.**

Environment-Specific Bean Configuration: In addition to profile-specific properties, you can use @ConditionalOnProperty and other conditional annotations to configure beans based on specific property values. This allows you to fine-tune the bean configuration based on different profiles or environment specific properties.



- **Test-Specific Configuration:** Profiles are also useful in test scenarios. You can define a test-specific profile (e.g., "test") and provide a separate set of configuration properties for testing purposes. This allows you to override or customize certain settings for testing without affecting other environments.
- By leveraging profiles, you can achieve a **clean separation of concerns, simplify configuration management,** and make your application more adaptable to different environments and use cases. Profiles enable you to easily **switch between configurations without modifying the code,** promoting greater flexibility and maintainability in your Spring Boot application.

Test Specific Configuration: Profiles are also useful in test scenarios. You can define a test-specific profile(eg"test") and provide a separate set of configuration properties for testing purposes. This allows you to override or customize certain settings for testing without affecting other environments.

By leveraging profiles, you can achieve a **clean separation of concerns, simply configuration management**, and make your application more adaptable to different environments and use cases. Profiles enable you to easily **switch between configurations without modifying the code,** promoting greater flexibility and maintainability in your spring Boot application.

Source : https://www.youtube.com/watch?v=cepbrLI0N20