Top 50 Most Frequent Java Interview Questions

This comprehensive guide presents the top 50 most frequently asked questions in Java interviews, curated from various online resources and expert opinions. Whether you're a fresher just starting your career or an experienced developer aiming for a senior role, this compilation will equip you with the knowledge and confidence to excel in your next Java interview.

Basic Concepts

Core Concepts

1. **What is Java?** Java is a high-level, object-oriented programming language known for its platform independence, security, and robustness. It's widely used for developing various applications, including web, mobile, and enterprise solutions.

2. **What is Java's 'write once, run anywhere' philosophy?** This philosophy refers to Java's ability to run on any platform with a Java Virtual Machine (JVM) without needing any code modifications. This is achieved through Java's bytecode, which can be executed on any system with a JVM.

3. **Explain JVM, JRE, and JDK.**
   - **JVM (Java Virtual Machine):** An abstract machine that provides a runtime environment for executing Java bytecode. It's responsible for platform independence.
   - **JRE (Java Runtime Environment):** Includes the JVM, core libraries, and other components necessary to run Java applications.
   - **JDK (Java Development Kit):** Provides tools for developing Java applications, including a compiler, debugger, and JRE.

4. **What is a JIT compiler?** A JIT (Just-In-Time) compiler improves the performance of Java programs by compiling bytecode into native machine code at runtime. This allows frequently executed code to run faster, as it doesn't need to be interpreted repeatedly.

Object-Oriented Principles

5. **What are the key features of Java?**

   Platform independence: Java code can run on any platform with a Java Virtual Machine (JVM), thanks to its "write once, run anywhere" principle.

   - **Object-oriented:** Java follows the principles of OOP, including encapsulation, inheritance, and polymorphism, promoting modularity and code reusability.

- **Robust:** Java has strong memory management and exception handling mechanisms, reducing errors and enhancing reliability.
- **Secure:** Java's security features protect against malicious code and unauthorized access.

6. **What are the differences between C++ and Java?**
   - **Memory management:** Java uses automatic garbage collection, while C++ requires manual memory management.
   - **Pointers:** Java doesn't support pointers, while C++ does.
   - **Platform dependence:** C++ is platform-dependent, while Java is platform-independent.
   - **Multiple inheritance:** C++ supports multiple inheritance of classes, while Java only supports multiple inheritance of interfaces.

7. **Why is Java not a "pure" object-oriented language?** Java is not considered pure object-oriented because it uses primitive data types (like int, float, char) which are not objects.

8. **Explain the concept of inheritance in Java.** Inheritance allows a class (subclass) to inherit properties and methods from another class (superclass), promoting code reuse and establishing an "is-a" relationship.

9. **What is method overloading?** Method overloading allows multiple methods with the same name but different parameters in the same class. The methods must have the same name but different parameter lists, which can vary by Number of parameters, Data types of parameters & Order of parameters.

```
public class Calculator {
    // Method with two int parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three int parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method with double parameters
    public double add(double a, double b) {
        return a + b;
    }
```

```
}
```

The return type alone cannot be used to differentiate overloaded methods.This won't work:

```
public int add(int a, int b) { return a + b; }
public double add(int a, int b) { return a + b; } // Compilation error
```

Java uses the parameter list to determine which method to call at compile time. For example:

```
Calculator calc = new Calculator();
calc.add(5, 3);          // Calls the first method
calc.add(5, 3, 2);       // Calls the second method
calc.add(5.5, 3.5);      // Calls the third method
```

10. **What is method overriding?** Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This allows subclasses to customize the behavior of inherited methods.

```
// Parent class
public class Animal {
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

// Child class
public class Dog extends Animal {
    @Override  // This annotation is recommended but optional
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }
}

// Child class
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

```
Animal myDog = new Dog();
myDog.makeSound();  // Outputs: "Woof! Woof!"

Animal myCat = new Cat();
myCat.makeSound();  // Outputs: "Meow!"
```

Data Types and Variables

11. **What are the differences between primitive data types and objects in Java?**
    - **Memory allocation:** Primitive data types store values directly, while objects store references to memory locations.
    - **Default values:** Primitive types have default values (e.g., 0 for int), while objects default to null.
    - **Immutability:** Some primitive wrapper classes (like String) are immutable, while objects can be mutable.
12. **What is the difference between String, StringBuilder, and StringBuffer?**
    - **String:** Immutable, meaning its value cannot be changed after creation.
    - **StringBuilder:** Mutable and not thread-safe, suitable for single-threaded string manipulation.
    - **StringBuffer:** Mutable and thread-safe, ideal for concurrent string manipulation.
13. **What is autoboxing and unboxing?**
    - **Autoboxing:** Automatic conversion of primitive types to their corresponding wrapper classes (e.g., int to Integer).
    - **Unboxing:** Automatic conversion of wrapper classes to their corresponding primitive types (e.g., Integer to int).

Access Control

14. **What are access modifiers in Java?** Access modifiers control the visibility of classes, methods, and variables. They include: - **public:** Accessible from any class. - **protected:** Accessible within the same package and subclasses. - **default:** Accessible only within the same package. - **private:** Accessible only within the same class. For example, using private access modifiers for sensitive data members ensures that they can only be accessed and modified within the class itself, enhancing data security and encapsulation.

15. **What is the purpose of the static keyword in Java?** The static keyword is used to define members (methods or variables) that belong to the class itself rather than instances of the class. Static methods can be called without creating an object of the class.

16. **What is the purpose of the final keyword in Java?** The final keyword can be applied to variables, methods, and classes: - **Variable:** Makes the variable a constant, its value cannot

be changed. - **Method:** Prevents the method from being overridden in subclasses. - **Class:** Prevents the class from being inherited.

17. **What is an interface?** An interface defines a contract that classes can implement. It specifies a set of methods that implementing classes must provide.

18. **What is an abstract class?** An abstract class cannot be instantiated and may contain abstract methods (methods without implementation). It serves as a blueprint for subclasses.

19. **What is the difference between an abstract class and an interface?**
    - **Method implementation:** Abstract classes can have both abstract and concrete methods, while interfaces can only have abstract methods (before Java 8).
    - **Variable types:** Abstract classes can have instance variables, while interfaces can only have constants.
    - **Inheritance:** A class can extend only one abstract class, but it can implement multiple interfaces.

20. **What is a constructor?** A constructor is a special method used to initialize objects of a class. It has the same name as the class and is called when an object is created.

21. **What is a package in Java?** A package is a way to organize related classes and interfaces into a namespace, preventing naming conflicts and improving code maintainability.

22. **What is the difference between the program and the process?**
    - A **program** is a set of instructions written in a programming language, like Java. It is a static entity.
    - A **process** is an instance of a program that is being executed. It is a dynamic entity with its own memory space and resources.

Collections Framework

23. **What are Java Collections?** The Java Collections Framework provides a set of interfaces and classes for storing and manipulating groups of objects. It offers various data structures like lists, sets, and maps, each with its own characteristics and performance considerations.

24. **What is the difference between List, Set, and Map?**
    - **List:** An ordered collection that allows duplicate elements. Examples include ArrayList and LinkedList.
    - **Set:** An unordered collection that does not allow duplicate elements. Examples include HashSet and TreeSet.

- **Map:** A collection of key-value pairs, where each key is unique and maps to a value. Examples include HashMap and TreeMap.

25. **What are the differences between ArrayList and LinkedList?**
    - **Implementation:** ArrayList uses a dynamic array, while LinkedList uses a doubly linked list.
    - **Performance:** ArrayList offers faster random access, while LinkedList provides efficient insertion and deletion. The choice between ArrayList and LinkedList depends on the specific use case. If frequent random access is required, ArrayList is preferred. However, if insertion and deletion operations are more common, LinkedList is a better choice.

26. **What is the difference between HashMap and Hashtable?**
    - **Synchronization:** Hashtable is synchronized, while HashMap is not. This means Hashtable is thread-safe but may have lower performance in single-threaded environments.
    - **Null values:** HashMap allows one null key and multiple null values, while Hashtable doesn't allow null keys or values.

27. **How does HashSet work internally in Java?** HashSet internally uses a HashMap to store its elements. Each element is stored as a key in the HashMap, with a dummy object as its value.

28. **How do you sort a collection in Java?** Collections can be sorted using the Collections.sort() method for lists and the TreeSet class for sets. The sort() method can take a comparator as an argument to customize the sorting order.

29. **How do you print an array in Java?** Arrays can be printed using the Arrays.toString() method, which converts the array to a string representation.

30. **What is the default size of ArrayList and HashMap in Java?**
    - The default size of ArrayList is 10.
    - The default size of HashMap is 16.

a. ArrayList Default Size:

```
// Default initial capacity is 10
ArrayList<String> list = new ArrayList<>();
```

- Initial capacity is 10 when created with no-args constructor
- When elements are added beyond capacity, ArrayList grows by 50% of its current size
- You can specify initial capacity: `new ArrayList<>(20)`

- Example of growth

```
ArrayList<String> list = new ArrayList<>();
// Size = 10
// After adding 10 elements,
// the list will resize to 15 (10 + 10/2)
```

b. HashMap Default Size:

```
// Default initial capacity is 16 with load factor of 0.75
HashMap<String, Integer> map = new HashMap<>();
```

- Initial capacity is 16 buckets
- Default load factor is 0.75 (75% full)
- When size exceeds (capacity × load factor), HashMap is rehashed to 2× its size
- You can customize both: `new HashMap<>(32, 0.8f)`
- Example of growth:

```
HashMap<String, Integer> map = new HashMap<>();
// Size = 16
// After adding 12 elements (16 * 0.75 = 12),
// the map will resize to 32 buckets
```

Exception Handling

31. **What is an exception?** An exception is an abnormal event that disrupts the normal flow of a program. Exceptions can be caused by various factors, such as invalid user input, network issues, or resource limitations.

32. **What are checked and unchecked exceptions?**
    - **Checked exceptions:** Checked at compile time, requiring explicit handling using try-catch blocks or throws declarations. Examples include IOException and SQLException.
    - **Unchecked exceptions:** Occur at runtime and don't require explicit handling. These are typically caused by programming errors. Examples include NullPointerException and ArrayIndexOutOfBoundsException. Understanding the distinction between checked and unchecked exceptions is crucial for writing robust and maintainable Java code. Checked exceptions enforce compile-time checks, promoting better error handling and preventing runtime surprises.

33. **How do you handle exceptions in Java?** Exceptions are handled using try-catch blocks. The try block contains the code that might throw an exception, and the catch block handles the exception if it occurs. Multiple catch blocks can be used to handle different types of exceptions.

34. **What is the purpose of the finally block?** The finally block is used to execute code regardless of whether an exception is thrown or caught. It's often used to release resources, such as closing files or database connections, to prevent resource leaks.

Multithreading and Concurrency

35. **What is multithreading in Java?** Multithreading allows multiple threads of execution to run concurrently within a single program, improving performance and responsiveness. Each thread has its own execution path and can perform tasks independently.

36. **How do you create a thread in Java?** Threads can be created by: - Extending the Thread class and overriding its run() method. - Implementing the Runnable interface and providing a run() method implementation.

    a. By extending the Thread class:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed in the thread
        System.out.println("Thread is running");
    }
}

// Usage:
MyThread thread = new MyThread();
thread.start();  // Starts the thread
```

    b. By implementing the Runnable interface (preferred approach):

```
public class MyRunnable implements Runnable {
    @Override
```

```
    public void run() {
        // Code to be executed in the thread
        System.out.println("Thread is running");
    }
}

// Usage:
Thread thread = new Thread(new MyRunnable());
thread.start();
```

37. **Explain the difference between Runnable and Callable interfaces.**
    - **Runnable:** The run() method doesn't return a value.
    - **Callable:** The call() method returns a value. This is useful for tasks that need to produce a result.

38. **What is synchronization?** Synchronization ensures that only one thread can access a shared resource at a time, preventing data inconsistency. This is typically achieved using synchronized blocks or methods, which use locks to control access to the shared resource.

39. **What is a deadlock?** A deadlock occurs when two or more threads are blocked forever, each waiting for the other to release a resource. For example, if thread A holds lock 1 and is waiting for lock 2, while thread B holds lock 2 and is waiting for lock 1, a deadlock occurs.

40. **What is a race condition?** A race condition occurs when the behavior of a program depends on the unpredictable order of execution of multiple threads. This can lead to unexpected and erroneous results.

41. **What is thread starvation?** Thread starvation occurs when a thread is unable to gain regular access to shared resources and is therefore unable to make progress. This can happen if other threads have higher priority or hold the required resources for extended periods.

42. **What is the ExecutorService interface and how does it work?** The ExecutorService interface provides a mechanism for managing a pool of threads and submitting tasks for execution. It simplifies thread management and improves efficiency by reusing threads instead of creating new ones for each task.

43. **How will you create a daemon thread in Java?** A daemon thread is a low-priority thread that runs in the background and provides services to user threads. To create a daemon thread, you can use the setDaemon(true) method on a thread object before starting it.

44. **What is time slicing?** Time slicing is a technique used by operating systems to allocate CPU time to different threads. Each thread gets a small slice of CPU time, and the CPU switches between threads rapidly, giving the illusion of parallel execution.

45. **What is a thread schedule?** A thread scheduler is a component of the operating system that determines which thread should run next. It uses various algorithms to prioritize threads and ensure fair allocation of CPU time.

46. **What is ThreadGroup? Why do companies not use it anymore?** ThreadGroup is a class that allows grouping of threads for management purposes. However, it has limitations and is generally not recommended for modern concurrency management. More sophisticated mechanisms like ExecutorService and concurrent collections are preferred.

Advanced Concepts

47. **What is the difference between fail-fast and fail-safe iterators?**
    - **Fail-fast:** Throw an exception (ConcurrentModificationException) if a collection is modified while iterating. This helps to detect concurrent modifications and prevent data inconsistency.
    - **Fail-safe:** Create a copy of the collection for iteration, allowing modifications to the original collection without affecting the iterator. This provides a more robust iteration mechanism but may have higher memory overhead.

48. **What is serialization?** Serialization is the process of converting an object into a byte stream, which can be stored in a file or transmitted over a network. This allows objects to be persisted and reconstructed later.

49. **What is the purpose of the transient keyword?** The transient keyword prevents a variable from being serialized. This is useful for fields that should not be persisted, such as sensitive data or temporary variables.

50. **What is reflection?** Reflection allows inspection and manipulation of classes, methods, and fields at runtime. This can be used for tasks like dynamically loading classes, invoking methods, and accessing private members.

51. **What are Java annotations?** Annotations provide metadata about code. They can be used for documentation, compile-time checks, and runtime processing. Examples include @Override and @Deprecated.

52. **What is a lambda expression?** A lambda expression is a concise way to represent an anonymous function. It's a key feature of Java 8 and functional programming. Lambda expressions allow you to write code that is more compact and expressive.

53. **What are Java Streams?** Streams provide a functional approach to processing collections of data. They allow operations like filtering, mapping, and reducing to be performed efficiently. Streams can improve code readability and performance, especially for large datasets.

54. **What is the Optional class?** The Optional class helps to avoid null pointer exceptions by providing a container for optional values. It encourages developers to explicitly handle the case where a value might be absent.

55. **What is a functional interface?** A functional interface is an interface with exactly one abstract method. It can be used with lambda expressions. Examples include Runnable and Comparator.

56. **What is the difference between sleep() and wait() methods?**
    - sleep() pauses the current thread for a specified time, but it doesn't release the lock. It is often used for introducing pauses or delays in a thread's execution, such as in polling scenarios.
    - wait() releases the lock on the object and waits for another thread to notify it. It is used for inter-thread communication, where one thread might wait for another thread to complete a task or signal an event.
57. **What is garbage collection in Java?** Garbage collection is the automatic process of reclaiming memory occupied by objects that are no longer in use. This relieves developers from manual memory management and reduces the risk of memory leaks. Java's automatic garbage collection simplifies memory management for developers, but it's essential to understand its mechanisms and potential performance implications, especially in resource-intensive applications.

58. **What is the difference between Heap and Stack Memory in Java?**
    - **Heap:** Stores objects and their instance variables. It is a shared memory area used by all parts of the application.
    - **Stack:** Stores method calls and local variables. Each thread has its own stack. Java utilizes heap memory for dynamic memory allocation of objects, while stack memory is used for managing method execution and local variables within each thread.

59. **What is a marker interface?** A marker interface is an empty interface that provides metadata about a class. It does not have any methods. Examples include Serializable and Cloneable.

60. **What is the JDBC API?** JDBC (Java Database Connectivity) is an API for connecting to and interacting with databases. It allows Java applications to execute SQL queries, retrieve data, and update databases.

61. **What is the purpose of the volatile keyword?** The volatile keyword ensures that a variable is read from and written to main memory, preventing caching issues in multithreaded environments. This is important for variables that are shared between multiple threads.

62. **Why would it be pointless for a static or final method to use dynamic binding?** Dynamic binding (also known as late binding) is a mechanism where the method to be called is determined at runtime. However, static and final methods are resolved at compile time, so dynamic binding wouldn't apply to them.

63. **How is a code point related to a code unit in Unicode?** In Unicode, a code point is a unique numeric value that represents a character. A code unit is a bit sequence used to encode code points. Different encoding schemes (like UTF-8 and UTF-16) use different code unit sizes.

64. **Can you tell us which three steps you would use to simulate a static class in Java?**

    Here are the three steps to simulate a static class in Java

```java
public final class MathUtils {  // Step 1: Make class final

    private MathUtils() {       // Step 2: Private constructor
        // Prevents instantiation
    }

    // Step 3: All static members
    public static int add(int a, int b) {
        return a + b;
    }

    public static double multiply(double a, double b) {
```

```
            return a * b;
        }
    }
```

Let's break down each step:

a. Declare the class as `final`
   - This prevents the class from being inherited
   - Ensures no one can extend and create instances of child classes

b. Create a private constructor
   - Prevents instantiation of the class from outside
   - Makes it impossible to create objects of this class

c. Make all methods and variables static
   - Allows methods to be called without creating an instance
   - Accessed directly through the class name: `MathUtils.add(5, 3)`

`Usage example:`

```
// This works:

int sum = MathUtils.add(5, 3);

// These won't work:

MathUtils utils = new MathUtils();  // Compilation error: private constructor

class MyMath extends MathUtils {}   // Compilation error: cannot inherit from
final class
```

65. **How will you use BlockingQueue to implement a producer-consumer problem?** BlockingQueue is a thread-safe queue that can be used to implement the producer-consumer pattern. Producers add items to the queue, and consumers take items from the queue. The blocking nature of the queue ensures that producers wait if the queue is full and consumers wait if the queue is empty.

66. **What are the ways to detect memory leaks in an application?** Memory leaks occur when objects are no longer needed but are still reachable, preventing garbage collection. Tools like profilers and heap analyzers can be used to detect memory leaks by identifying objects that are consuming excessive memory.

Top 50 Spring Boot Interview Questions and Answers

Spring Boot has revolutionized the way Java developers build and deploy applications. Its popularity has made it a common topic in technical interviews, especially for those seeking roles involving Java and backend development. This article presents a comprehensive list of the top 50 most frequently asked Spring Boot interview questions, categorized by experience level and covering a wide range of essential concepts. Whether you're a fresher just starting your journey or an experienced developer looking to brush up on your knowledge, this guide will equip you with the insights needed to ace your next Spring Boot interview.

Spring Boot Fundamentals

These questions assess your basic understanding of Spring Boot and its core principles:

1. **What is Spring Boot?**

   Spring Boot is an open-source Java-based framework used to create stand-alone, production-ready Spring applications. It simplifies Spring application development by reducing boilerplate code, providing auto-configuration for common components, and offering a range of production-ready features.

2. **What are the advantages of using Spring Boot?**

   Spring Boot offers several advantages, including:

   - **Faster development and deployment:** Spring Boot simplifies the setup and development process, allowing for quicker turnaround times.
   - **Reduced boilerplate code:** Spring Boot automates many configuration tasks, reducing the amount of code you need to write.
   - **Simplified dependency management:** Spring Boot provides starter dependencies that manage common libraries and versions, simplifying dependency management.
   - **Auto-configuration for common scenarios:** Spring Boot automatically configures common components and settings based on your project's dependencies.
   - **Embedded servers for easy testing and deployment:** Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, making it easy to test and deploy applications without needing external servers.
   - **Production-ready features:** Spring Boot includes features like health checks, metrics, and monitoring to help you manage and monitor your applications in production.

3. **Why is Spring Boot preferred over Spring?**

   Spring Boot is preferred over traditional Spring because it eliminates boilerplate configuration through automatic configuration, creates standalone applications with embedded servers, provides production-ready features like metrics and health checks out

of the box, and enables faster development with tools like Spring Initializr and starter POMs. Unlike traditional Spring, which requires extensive manual configuration, Spring Boot follows "convention over configuration" principle, making it simpler to create and deploy applications while still maintaining Spring's core features.

4. **What are the key components of Spring Boot?**

   Spring Boot's key components include:

   - **Spring Boot Starter:** Provides opinionated dependencies to simplify build configuration.
   - **Spring Boot AutoConfigurator:** Automatically configures Spring applications based on the classpath.
   - **Spring Boot Actuator:** Offers production-ready features like health checks and metrics.
   - **Embedded HTTP Servers:** Includes embedded servers like Tomcat, Jetty, and Undertow.
   - **Spring Boot CLI:** Provides a command-line interface for creating and managing Spring Boot applications.

5. **What are the Spring Boot Starter Dependencies?**

   Starter dependencies are pre-configured sets of dependencies that simplify dependency management. Some commonly used starters include:

   - spring-boot-starter-web: For building web applications.
   - spring-boot-starter-data-jpa: For using Spring Data JPA.
   - spring-boot-starter-security: For adding security features.
   - spring-boot-starter-test: For testing Spring Boot applications.
   - spring-boot-starter-thymeleaf: For using Thymeleaf template engine.
   - spring-boot-maven-plugin: Provides support for building and packaging Spring Boot applications with Maven.

6. **How does Spring Boot handle application events?**

   Spring Boot utilizes application events and listeners to handle various tasks and enable asynchronous communication between different parts of your application. You can create custom events and listeners to respond to specific events within your application.

Annotations in Spring Boot

These questions focus on your understanding of key annotations used in Spring Boot:

7. **What is the purpose of @SpringBootApplication?**

   @SpringBootApplication is a composite annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It marks the main class of a Spring Boot application, enabling auto-configuration, component scanning, and defining the application context.

8. **What does the @EnableAutoConfiguration annotation do?**

   @EnableAutoConfiguration enables Spring Boot's auto-configuration mechanism. It automatically configures beans based on the classpath and other application settings.

9. **What is the purpose of using @ComponentScan in the class files?**

   @ComponentScan instructs Spring to scan the specified packages for components (e.g., controllers, services, repositories) and register them as beans in the application context.

10. **What is the difference between @RestController and @Controller in Spring Boot?**
    - @Controller is used for traditional Spring MVC applications where you return a view name.
    - @RestController is a specialized version of @Controller that combines @Controller and @ResponseBody, making it suitable for building RESTful web services that return data directly.

11. **What is the difference between @Component, @Service, and @Repository annotations?**

    These annotations are specializations of @Component:

    - @Component: A generic stereotype for any Spring-managed component.
    - @Service: A specialization for service layer classes.
    - @Repository: A specialization for persistence layer classes (DAOs).

Spring Boot Configuration

These questions delve into how Spring Boot handles configuration and externalized properties:

12. **How does Spring Boot handle externalized configuration?**

    Spring Boot supports externalized configuration through various sources, including:

    - Properties files (e.g., application.properties, application.yml)
    - YAML files
    - Environment variables
    - Command-line arguments

13. **Describe the role of application.properties and application.yml.**

    application.properties and application.yml are configuration files used in Spring Boot to externalize application settings. They allow you to define properties like database connection details, server port, and logging configurations.

14. **How can you handle different environments in a Spring Boot application?**

    Spring profiles allow you to define environment-specific configurations. You can create separate property files (e.g., application-dev.properties, application-prod.properties) and activate a profile using spring.profiles.active.

Spring Boot Actuator

These questions explore the capabilities of Spring Boot Actuator:

15. **What is Spring Boot Actuator?** Spring Boot Actuator provides production-ready features for monitoring and managing your Spring Boot application. It offers endpoints that expose information about the application's health, metrics, environment, and configuration.
16. **What are the actuator-provided endpoints used for monitoring the Spring boot application?** Actuator endpoints provide insights into various aspects of the application:
    - /health: Shows the health status of the application.
    - /metrics: Provides various metrics about the application, such as memory usage and request counts.
    - /env: Displays environment properties.
    - /info: Shows custom application information.
    - /loggers: Allows you to view and modify logging levels.

17. **How to enable Actuator in Spring boot application?** Add the spring-boot-starter-actuator dependency to your project's pom.xml or build.gradle file.

Dependency Injection

These questions test your understanding of dependency injection in Spring Boot:

18. **What is Dependency Injection in Spring Boot?**

    Dependency Injection (DI) is a design pattern where objects are not responsible for creating their own dependencies. Instead, dependencies are "injected" into objects by a container (Spring IoC container). This promotes loose coupling and makes code more testable and maintainable.

19. **What is an IOC container?**

    An IoC (Inversion of Control) container is a core part of the Spring framework. It manages the creation, configuration, and lifecycle of beans (objects) in your application.

20. **What is the difference between Constructor and Setter Injection?**

    Constructor injection provides dependencies through the constructor, making them required and immutable. Setter injection provides dependencies via setter methods after object

creation, making them optional and mutable. Constructor injection is preferred for mandatory dependencies, while setter injection suits optional or changeable dependencies.

Data Access

These questions cover Spring Boot's data access features:

21. **How does Spring Boot support database operations?**

    Spring Boot provides excellent support for database operations through Spring Data JPA, which simplifies database interactions and reduces boilerplate code. To connect to a database using Spring Data JPA, you need to:

    - Add the spring-boot-starter-data-jpa dependency to your project.
    - Configure the database connection details in your application.properties or application.yml file.
    - Define JPA entities and repositories to interact with your database.

22. **How do you configure multiple data sources in Spring Boot?**

    To configure multiple data sources in Spring Boot, you need to define each data source in the application.properties or application.yml file, create separate configuration classes for each data source using @Configuration and @Primary annotations, and configure LocalContainerEntityManagerFactoryBean and PlatformTransactionManager beans for each data source.

Templating Engines

This section explores templating engines used in Spring Boot applications:

23. **What is Thymeleaf? How do you use it in a Spring Boot application?**

    Thymeleaf is a popular server-side Java template engine for web and standalone environments. To use Thymeleaf in a Spring Boot application, you should include the spring-boot-starter-thymeleaf dependency in your project. This will automatically configure Thymeleaf and make it ready to use for creating dynamic web pages.

REST Controllers and API Development

These questions focus on building RESTful APIs with Spring Boot:

24. **What is @RequestMapping annotation in Spring Boot?**

    @RequestMapping maps web requests to specific handler methods in controllers. It can be used at the class level to define a base mapping and at the method level to define specific request mappings.

25. **What is the difference between RequestMapping and GetMapping?**
    - @RequestMapping is a general-purpose annotation for mapping requests with different HTTP methods (GET, POST, PUT, DELETE, etc.).
    - @GetMapping is a specialized version of @RequestMapping specifically for handling GET requests.

26. **Have you implemented REST API versioning? What was the need, and how did you implement it? What are the possible ways to implement versioning in a Spring Boot application?**

    API versioning is important for maintaining backward compatibility while evolving your API. There are several ways to implement versioning in Spring Boot, such as using URL path versioning (e.g., /v1/users), request parameter versioning (e.g., /users?version=1), or header versioning (e.g., Accept: application/vnd.myapp.v1+json).

Exception Handling

These questions assess your knowledge of exception handling in Spring Boot:

27. **How do you handle exceptions in Spring Boot?**

    Spring Boot provides several ways to handle exceptions:

    - Using @ControllerAdvice to define global exception handlers.
    - Using @ExceptionHandler to handle specific exceptions in controllers.
    - Using ResponseEntityExceptionHandler to customize the response for standard Spring exceptions.

Spring Boot DevTools

This question explores the developer-friendly features of Spring Boot DevTools:

28. **What is Spring Boot DevTools?**

    Spring Boot DevTools is a module that provides developer-friendly features like automatic restarts, live reload, and remote debugging to improve developer productivity.

Logging

This question covers Spring Boot's logging capabilities:

29. **How does Spring Boot support logging?**

Spring Boot uses Commons Logging for internal logging and provides default configurations for popular logging libraries like Logback, Log4j2, and Java Util Logging. You can configure logging levels, output formats, and appenders in external configuration files.

Embedded Servers

This question delves into Spring Boot's embedded server support:

30. **What is Spring Boot's embedded server, and how do you configure it?**

    Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow. You can configure the server by setting properties in application.properties or application.yml, such as the server port (server.port). The default port number for a Spring Boot application is 8080, but it can be changed in the application's configuration file (e.g., application.properties or application.yml) using the property server.port[11]. Embedded servers eliminate the need to set up and manage external application servers, simplifying development and testing.

Spring Boot CLI

These questions assess your familiarity with the Spring Boot CLI:

31. **What is Spring Boot CLI and what are its benefits?**

    Spring Boot CLI is a command-line interface for creating and managing Spring Boot applications. It offers features like:

    - Auto-dependency resolution
    - Auto-configuration
    - Groovy scripting support
    - No semicolons
    - No public/private access modifiers
    - No return statements (in most cases)

32. **What are the most common Spring Boot CLI commands?**

    Some common CLI commands include:

    - init: Creates a new Spring Boot project.
    - run: Runs a Spring Boot application.
    - test: Runs tests for a Spring Boot application.

Spring Initializr

This question covers the role of Spring Initializr in bootstrapping Spring Boot projects:

33. **What is Spring Initializr?**

    Spring Initializr is a web-based tool and API that simplifies the creation of Spring Boot projects. It allows you to select dependencies, project metadata, and generate a project structure.

Miscellaneous

These questions cover various other aspects of Spring Boot:

34. **What are the different scopes available in Spring Boot?**

    Spring beans can have different scopes:

    - **singleton**: A single instance of the bean is created per application context.
    - **prototype**: A new instance of the bean is created each time it is injected.
    - **request**: A new instance of the bean is created for each HTTP request.
    - **session**: A new instance of the bean is created for each user session.
    - **application**: A single instance of the bean is created per ServletContext.

35. **How to get the list of all the beans in your Spring boot application?**

    You can use the listBeans() method of the ApplicationContext to get a list of all beans defined in your Spring Boot application.

36. **Can we disable the default web server in the Spring Boot application?**

    Yes, you can disable the default web server by setting webApplicationType to NONE in the @SpringBootApplication annotation.

37. **How to disable a specific auto-configuration class?**

    You can disable specific auto-configuration classes using the exclude attribute of @EnableAutoConfiguration.

Spring Boot for Experienced Developers

These questions are geared towards experienced developers and delve into more advanced topics:

38. **How can you check the environment properties in a Spring Boot application?**

    You can access environment properties using the Environment object or by injecting

properties with @Value.

39. **Can you please explain Spring Boot Dependency Management?**

    Spring Boot manages dependencies through starter POMs, which define common dependencies and their versions. This simplifies dependency management and ensures consistency across projects.

40. **What is the procedure for creating a non-web application in Spring Boot?**

    To create a non-web application, exclude the spring-boot-starter-web dependency and use appropriate dependencies for your application type (e.g., spring-boot-starter-batch for batch processing).

41. **What is meant by the Spring Boot Annotations?**

    Spring Boot uses annotations to simplify configuration and provide metadata to the framework. Some common annotations include @SpringBootApplication, @RestController, @Service, @Repository, and @EnableAutoConfiguration.

42. **Describe the idea of externalized logging utilizing either Logback or Log4j2.**

    Spring Boot supports externalized logging configuration using Logback or Log4j2. You can configure logging levels, output formats, and appenders in external configuration files.

43. **State the differences between Spring MVC and Spring Boot.**

    Spring MVC is a framework for building web applications, while Spring Boot builds on top of Spring MVC to provide a more streamlined and opinionated approach to development.

44. **State the differences between Spring Data JPA Hibernate.**

    Spring Data JPA is an abstraction layer that simplifies database interactions, while Hibernate is a specific implementation of JPA.

45. **What are conditional annotations? Have you used any such annotations?**

    Conditional annotations allow you to conditionally enable or disable components or configurations based on certain conditions. Examples include @ConditionalOnProperty, @ConditionalOnBean, and @ConditionalOnMissingBean.

46. **What is the use of @EnableAutoConfiguartion annotation in spring boot? How does spring boot achieve auto configuration?**

    @EnableAutoConfiguration triggers Spring Boot's auto-configuration mechanism, which automatically configures beans based on the classpath and other application settings.

47. **If we do not want a dependency to be auto-configured by AutoConfiguration, what steps do we need to take?**

    You can exclude specific auto-configuration classes using the exclude attribute of @EnableAutoConfiguration or by defining your own configuration that overrides the auto-configured beans.

48. **How do you monitor your spring boot application in production? Which dependency you use?**

    You can monitor Spring Boot applications using tools like Spring Boot Actuator, Micrometer, and Prometheus. The spring-boot-starter-actuator dependency provides basic monitoring endpoints.

49. **What are the ways to secure Apis? Which all starter dependencies you have used in application?**

    You can secure APIs using Spring Security, which provides authentication, authorization, and other security features. The spring-boot-starter-security dependency is used to integrate Spring Security.

50. **What is rate limiting? Have you used one?**

    Rate limiting is a technique to control the rate of requests to an API or service. It helps prevent abuse, protect against denial-of-service attacks, and ensure fair usage. Spring Boot applications can implement rate limiting using libraries like Bucket4j or Resilience4j.

51. **What are interceptors and How are you using it in your spring boot application? Explain life cycle of interceptors (different phases involved)**

    Interceptors intercept requests before and after they reach the controller. They can be used for tasks like logging, security checks, and modifying request/response data. Interceptors have three main phases: preHandle (before the handler method is invoked), postHandle (after the handler method completes), and afterCompletion (after the complete request is finished).

Types of Spring Boot Interview Questions

In addition to the technical questions covered above, be prepared for different types of questions in a Spring Boot interview:

- **Fundamentals:** These assess your basic understanding of Spring Boot concepts, annotations, and configurations.
- **Advanced Topics:** These delve into more complex areas like security, performance optimization, and integration with other technologies.

- **Problem-Solving:** These evaluate your ability to apply Spring Boot knowledge to solve real-world problems. You might be asked to design solutions or troubleshoot issues.
- **Coding Exercises:** These test your coding skills and ability to implement Spring Boot features.

Remember to demonstrate your problem-solving skills, your ability to apply Spring Boot concepts to real-world scenarios, and your passion for continuous learning. Good luck!