Java8

1.Difference between Runnable and Callable interface

=====================================================

Real-World Analogy:

Imagine you’re ordering food through a delivery app like Swiggy or Zomato:

1)Runnable (Fire-and-forget): You place a simple order for food, but you don’t expect the app to give you any extra details like the chef's name or the estimated preparation time.

Once the order is placed, you can just wait for it to arrive. The app (thread) executes the task without returning any additional result.

2)Callable (Result-oriented): You place a more detailed order where you want extra information—like the exact preparation time, delivery ETA, or discounts applied. Here,

the app (thread) not only processes the order but also returns these additional details as a result.

In programming terms:

1)Runnable is like "just do the task."

2)Callable is like "do the task and return the result."



2.Features of java8

===================

1. Lambda Expressions

A concise way to represent anonymous functions.

Simplifies the implementation of functional interfaces.

2. Functional Interfaces

Interfaces with a single abstract method (SAM), used with Lambda expressions.

Examples: Runnable, Callable, and new ones like Predicate, Function, and Consumer.

Annotated with @FunctionalInterface for clarity.

3. Streams API

Enables functional-style operations on collections and sequences of data.

Simplifies operations like filtering, mapping, and reducing large data sets.

4. Default Methods in Interfaces

Allows interfaces to have methods with implementation, using the default keyword.

Enables backward compatibility without breaking existing code.

5. Optional Class

A container object to handle null values more gracefully.

Reduces NullPointerException by representing optional values explicitly.

6. Date and Time API (java.time)

A modern, thread-safe Date and Time API replacing java.util.Date.

Classes like LocalDate, LocalTime, LocalDateTime, and ZonedDateTime introduced.

7. Stream Enhancements in Collections

Collections like List, Map, etc., gained new methods (e.g., forEach, stream, parallelStream).

8. Nashorn JavaScript Engine

A new JavaScript engine that replaces Rhino.

Enables integration of JavaScript code within Java applications.

9. Parallel Array Sorting

Faster sorting of arrays using the Arrays.parallelSort() method.

10. Base64 Encoding and Decoding

Built-in support for Base64 encoding and decoding in the java.util.Base64 class.

3. What are intermediate and terminal operater in stream

==========================================================

Intermediate Operations

These operations transform a Stream into another Stream.

They are lazy—they don't execute until a terminal operation is invoked.

Multiple intermediate operations can be chained together.

Examples:

filter(): Filters elements based on a condition.

map(): Transforms each element.

sorted(): Sorts elements.

distinct(): Removes duplicates.

Terminal Operations

These operations produce a result from the Stream (e.g., a collection, a value, or a side effect).

They trigger the execution of intermediate operations.

Examples:

forEach(): Performs an action on each element.

collect(): Converts Stream into a collection (e.g., List, Set).

reduce(): Reduces elements to a single result.

count(): Counts the number of elements.

4. What is method refrences in java8

====================================

In Java 8, Method References are a shorthand notation of Lambda expressions to call methods directly by referring to them. They improve code readability and reusability by

avoiding boilerplate code. Method references can be used where a Lambda Expression is used, as long as the method signature matches the functional interface.

5. What is Predicate, Function, Consumer and Supplier interface

================================================================

1. Predicate (Takes input, returns a boolean)

Used to evaluate a condition or predicate on an input.

Method: boolean test(T t)

Example:

Predicate<Integer> isEven = num -> num % 2 == 0;

System.out.println(isEven.test(4)); // Output: true

Real-Life Use Case: Filtering a list of numbers to find all even numbers.

2. Function (Takes input, returns a result)

Used to transform data from one form to another.

Method: R apply(T t)

Example:

Function<String, Integer> lengthFunction = str -> str.length();

System.out.println(lengthFunction.apply("Hello")); // Output: 5

Real-Life Use Case: Converting a list of names into their lengths.

3. Consumer (Takes input, returns nothing)

Used to perform operations on an input without returning anything.

Method: void accept(T t)

Example:

Consumer<String> printer = message -> System.out.println(message);

printer.accept("Hello, Consumer!"); // Output: Hello, Consumer!

Real-Life Use Case: Printing elements of a list or logging messages.

4. Supplier (Takes no input, returns a result)

Used to supply values, often as a factory or lazy initialization.

Method: T get()

Example:

Supplier<Double> randomSupplier = () -> Math.random();

System.out.println(randomSupplier.get()); // Output: Random number

6. What is double collan operator in java 8

===========================================

The double colon operator (::) in Java 8 is used for method references. It provides a shorthand way to refer to methods or constructors without having to invoke them directly.

Essentially, it helps to make your Lambda expressions more concise and readable.

How It Works:

The :: operator can be used wherever a functional interface is expected, and the method reference matches the interface's single abstract method (SAM).

7. What are the changes in HashMap from Java 8 onwards.

========================================================

From Java 8 onwards, HashMap received some key improvements and optimizations to enhance performance and reduce memory overhead, particularly in scenarios involving large datasets or

hash collisions. Here's an overview of the changes:

1. Introduction of TreeNodes for Buckets (TreeifyBins)

Before Java 8: HashMap used a linked list to store multiple entries in a bucket when hash collisions occurred. If many keys mapped to the same bucket, the linked list would degrade performance to O(n) in the

worst case.

From Java 8: If the number of entries in a bucket exceeds a threshold (8 by default), the linked list is replaced with a balanced binary search tree (TreeNode). This improves performance in such cases to O(log n).

2. Dynamic Resizing Optimization

Java 8 includes better resizing mechanisms when the HashMap grows. During resizing, the tree-based structure and hashing are more efficient compared to earlier versions.

3. Improved Hash Function

Java 8 introduced a more efficient and high-quality hash function (hash() method) to reduce the chances of hash collisions. This spreads keys more evenly across buckets.

4. Performance Optimization with compute() Methods

Java 8 added several new methods for concurrent computation and updates in the map, such as:

compute()

computeIfAbsent()

computeIfPresent() These methods allow atomic updates and reduce the need for external synchronization.

5. Default Methods from Map Interface

Java 8 added default methods to the Map interface, which are inherited by HashMap. Examples include:

forEach(BiConsumer action)

replaceAll(BiFunction function)

merge(K key, V value, BiFunction function)

Summary of Benefits

Improved Collision Handling: Tree-based buckets reduce performance degradation.

Faster Resizing: Better handling during capacity expansion.

Enhanced API: New methods for dynamic computation and streamlined operations.

Better Hashing: More uniform key distribution.

8. What changes is done in garbage collection after java8

**🚀 Main Changes in Garbage Collection After Java 8:**

1. **G1 GC is Default (Java 9 onwards)**
   * Replaced the old default (Parallel GC).
   * Faster and smoother with shorter pause times.
2. **New Garbage Collectors Introduced:**
   * **ZGC (Java 11):** Very low pause times, good for large memory apps.
   * **Shenandoah (Java 12):** Also low-pause GC, similar to ZGC.
   * **Epsilon (Java 11):** No garbage collection at all (for testing only).
3. **Old CMS GC Removed:**
   * CMS was removed in Java 14.
   * G1 or new GCs are recommended instead.
4. **Better GC Logs and Monitoring (Java 9+):**
   * Easier to read logs using -Xlog.
5. **Faster Full GC in G1 (Java 11):**
   * G1’s full GC became parallel and quicker.

✅ **In short:**  
Java made garbage collection faster, added new modern GCs, removed the old CMS collector, and improved logging and performance after Java 8.

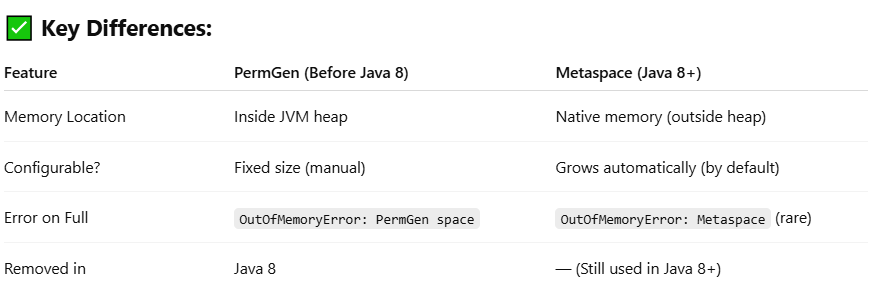
9) What is perm gen and metaspace in java8

**🔸 What is PermGen (Before Java 8)?**

* **PermGen (Permanent Generation)** was a memory area in the JVM.
* It stored **class metadata**, like:
  + Class names
  + Method names
  + Static variables
* Had a **fixed size**, set with -XX:PermSize and -XX:MaxPermSize.
* If it got full, you’d get an error:  
  **java.lang.OutOfMemoryError: PermGen space**

**🔸 What is Metaspace (Java 8 and Later)?**

* **PermGen was removed in Java 8**, replaced by **Metaspace**.
* **Metaspace stores class metadata**, just like PermGen.
* But the big improvement:
  + **It grows automatically** as needed (by default).
  + Stored **in native memory** (not inside JVM heap).
* You can limit it using:
  + -XX:MaxMetaspaceSize
  + -XX:MetaspaceSize



10) Difference between Filter and Map

**🔹 filter()**

* **Used to select elements** that match a condition.
* It **removes** elements that don’t meet the condition.
* Output size is **less than or equal** to input size.

List<Integer> numbers = List.of(1, 2, 3, 4, 5);

List<Integer> evenNumbers = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

System.out.println(evenNumbers); // Output: [2, 4]

**🔹 map()**

* **Used to transform** each element.
* It **changes the data** but keeps the number of elements the same.
* Output size is **equal** to input size.

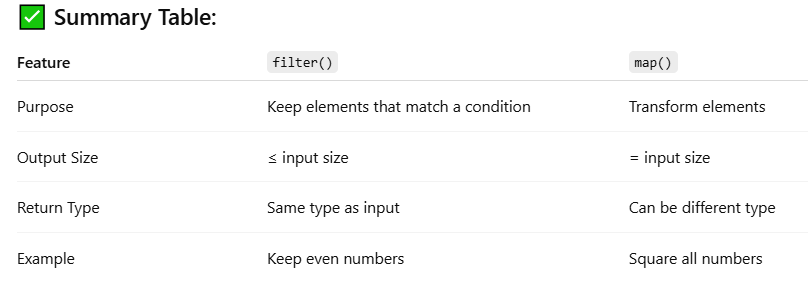
List<Integer> numbers = List.of(1, 2, 3);

List<Integer> squares = numbers.stream()

.map(n -> n \* n)

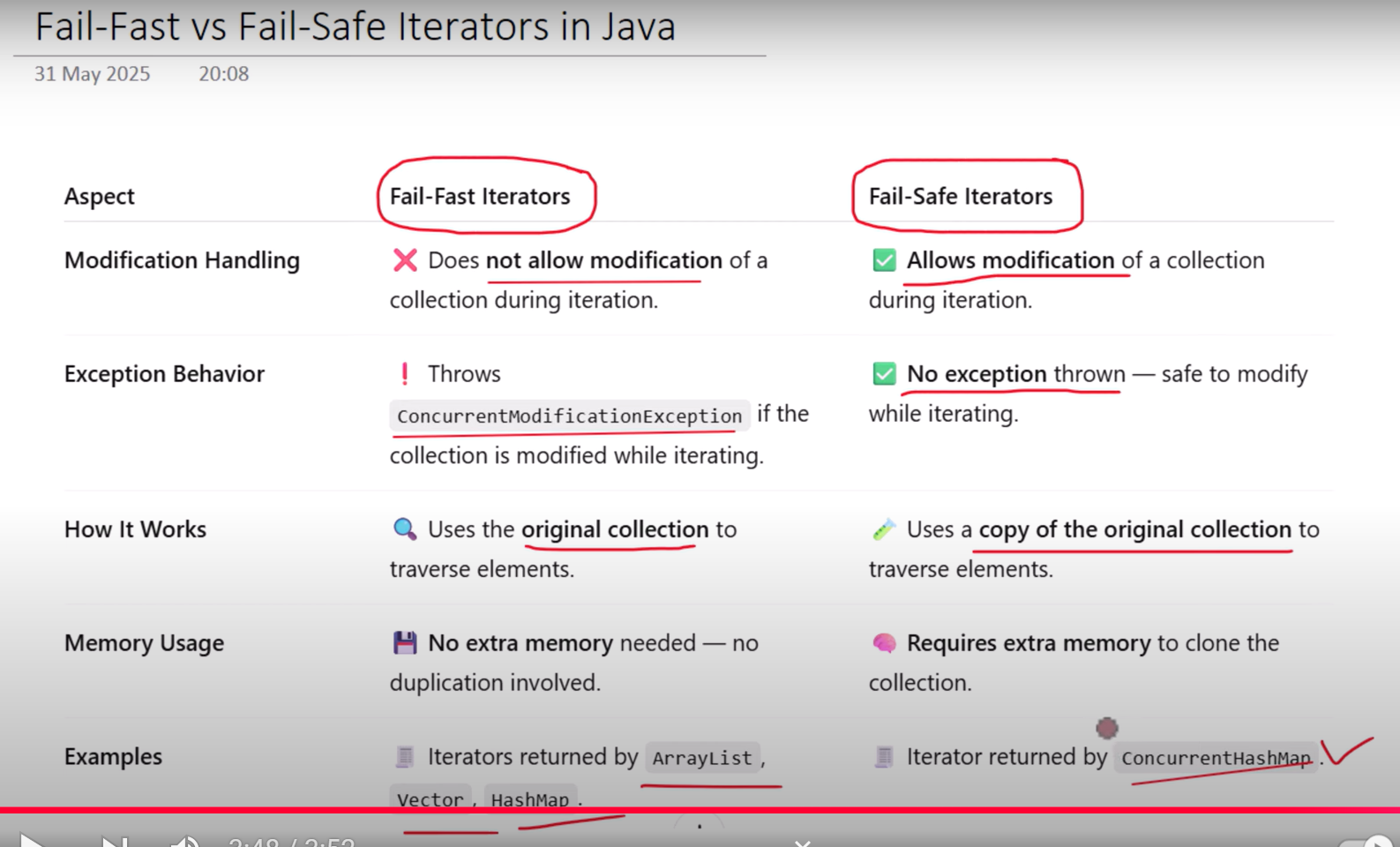
.collect(Collectors.toList());

System.out.println(squares); // Output: [1, 4, 9]



11) Can we extend functional interface ?

Yes



Q 13 javahungry

Collection

1. Use of Concurrent HashMap

============================

ConcurrentHashMap is a thread-safe map in Java designed for concurrent access without external synchronization. It allows multiple threads to read and write efficiently by

locking only specific segments of the map during updates. Reads are non-blocking, and it doesn't allow null keys or values. Commonly used for real-time data sharing,

caching, and counting in multithreaded applications.

2. What is HashCollision

==========================

A hash collision happens when two different keys generate the same hash value in a hash table, causing them to map to the same location. This is resolved using techniques like

chaining (storing multiple keys in a list for the same bucket) or open addressing (finding an alternate slot). Proper hash functions minimize collisions.

3. Difference between Comparable and Camparator (How we use comparator)



4. What is load factor of HashMap

==================================

The load factor in a HashMap is the measure of how full the map can get before resizing. It determines when to increase capacity to reduce collisions.

The default load factor is 0.75, meaning the map resizes when it's 75% full. This balances performance between minimizing collisions and avoiding frequent resizing.

5. Benefits of using generic

=============================

Generics in Java provide type safety, ensuring errors are caught at compile-time, eliminate explicit casting, making code cleaner, enable code reusability for

different data types, improve readability, and detect potential issues earlier. This leads to safer, more efficient, and modular code.

6. What is a use blocking queue ?

==================================

A BlockingQueue is a thread-safe queue in Java used in multithreading. It automatically blocks producer threads if the queue is full and consumer threads if the queue is empty.

It's commonly used in producer-consumer scenarios to simplify thread communication and ensure safe access without manual synchronization.

7. Internal working of concurrent HashMap.

==========================================

ConcurrentHashMap is a thread-safe map designed for high performance in multithreaded environments. It allows multiple threads to read and write simultaneously by locking only

specific buckets during modifications, ensuring minimal contention. Reads are non-blocking, and it does not allow null keys or values.

This ensures thread-safe and efficient operations.

Exception handling

1. What is class cast Exception

===================================

A ClassCastException in Java is thrown when an object is incompatibly cast from one class type to another. It occurs at runtime when you try to cast

an object to a subclass or an unrelated class without proper type compatibility.

2. What is checked and unchecked exception

=============================================

Checked Exceptions (Compile-Time)

Checked at compile time—must be handled using try-catch or throws.

Caused by external factors like file handling, database access, etc.

Examples: IOException, SQLException, FileNotFoundException.

Unchecked Exceptions (Runtime)

Not checked at compile time—handling is optional.

Caused by programming errors like null values, index out of bounds, etc.

Examples: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

3. What is the use of finally block.

====================================

The finally block is used to execute important cleanup code that must run regardless of whether an exception occurs or not.

It is mainly used for closing resources like files, database connections, or network sockets.

Multhithreading

1. What is Transient variable

A transient variable is a variable that is not serialized when an object is converted into a byte stream (serialization). This means the value of a

transient variable is not saved when writing an object to a file or sending it over a network.

Key Points About transient Keyword

✔ Prevents serialization of a variable.

✔ Used for sensitive data like passwords.

✔ Non-transient variables are serialized normally.

✔ Applicable only to instance variables (not static or local variables).

When to Use transient?

✔ Security Reasons – Avoid storing sensitive data like passwords.

✔ Large Data Fields – Prevent serialization of unnecessary large fields.

✔ Non-Persistent Data – Temporary values that don’t need to be saved.

What is a Byte Stream in Java?

A byte stream is a way of reading and writing data one byte at a time in Java. It is used for handling binary data like images, audio, videos,

and other non-text files.

2. What is Serialization and deserilization ?

==============================================

🔹 Serialization

Serialization is the process of converting an object into a byte stream so that it can be saved to a file, database, or sent over a network.

Used for storing objects or transferring data between systems.

Implemented using Serializable interface.

🔹 Deserialization

Deserialization is the process of converting a byte stream back into an object.

Used to retrieve saved objects from files, databases, or networks.

Key Points

✔ Uses ObjectOutputStream (for serialization) and ObjectInputStream (for deserialization).

✔ Transient variables are not serialized.

✔ Static variables are not serialized (because they belong to the class, not an object).

2. How many ways we can create thread

======================================

Extending Thread Class

📌 Steps:

Extend the Thread class.

Override the run() method.

Start the thread using .start().

✅ Easy to use, but Java doesn’t support multiple inheritance, so extending Thread prevents extending another class.

2️⃣ Implementing Runnable Interface

📌 Steps:

Implement Runnable.

Override run() method.

Pass an instance of Runnable to Thread.

Start using .start().

Better approach because it allows multiple inheritance (class can extend another class too).

Method Real-Life Example

Extending Thread Food Delivery – A single person picks up & delivers food.

Implementing Runnable Bank Transaction – A bank system handles multiple transactions at once.

3. What is thread life cycle

============================

🔹 Thread Life Cycle in Java (With Real-Time Example 🚀)

A thread in Java has 5 states in its life cycle:

1️⃣ New – Thread is created but not started.

2️⃣ Runnable – Ready to run but waiting for CPU.

3️⃣ Running – Actively executing.

4️⃣ Blocked/Waiting – Paused for a resource.

5️⃣ Terminated – Execution is complete.

Real-Time Example: Food Delivery App 🚴

📌 Scenario: A delivery agent delivering food in an app.

NEW → Order received.

RUNNABLE → Agent is ready but waiting for a bike.

RUNNING → Agent is delivering the order.

WAITING → Agent is waiting for traffic to clear.

TERMINATED → Order is delivered.

4. Difference between wait and sleep method

wait(): Customer is waiting for food. The chef will notify them when food is ready.

sleep(): Chef is taking a break for 5 minutes before preparing the next order.

5. what is join method

========================

🔹 What is join() in Java?

📌 The join() method is used in multithreading to pause the execution of one thread until another thread finishes its task.

🔹 Why Use It?

Ensures that a thread completes its execution before moving to the next step.

Useful when one thread depends on another.

🔹 Real-Time Example: Movie Ticket Booking 🎬

Imagine you are booking a movie ticket online:

1️⃣ Payment Thread (Thread-1) → First, the payment must be completed.

2️⃣ Ticket Generation Thread (Thread-2) → The ticket will only be generated after payment is done.

6 What is valatile keyword

===========================

🔹 Simple Explanation of volatile (With Real-Life Example)

Think of volatile like a live scoreboard in a cricket match 🏏.

The scoreboard shows the latest score visible to everyone (players, audience, umpires).

Whenever the score changes, everyone immediately sees the update because it's directly changed on the main display.

Now, imagine a banking system 🏦 where:

1️⃣ A customer is checking their transaction status.

2️⃣ The bank updates the status once the transaction is completed.

Without volatile:

The customer might still see the old status due to caching.

The update is not immediately visible to all threads.

With volatile:

The customer always sees the latest status because it's directly read from memory.

The volatile keyword in Java is used to ensure visibility and prevent thread caching issues for variables in multi-threaded environments.

✅ Key Features of volatile:

✔ Ensures latest value of a variable is read directly from main memory, not from CPU cache.

✔ Prevents thread caching issues in multi-threaded applications.

✔ Does not guarantee atomicity (for operations like x++, use synchronized or AtomicInteger).

7. Explain inter thread communication

======================================

🔹 What is Inter-Thread Communication in Java?

📌 Inter-Thread Communication is the process where multiple threads communicate with each other using methods like wait(), notify(), and notifyAll().

🔹 Why Use It?

When one thread needs to pause and wait for another thread’s task.

It helps in avoiding busy waiting and makes thread execution more efficient.

🔹 Real-Time Example: Restaurant Ordering System 🍽️

Imagine a restaurant where:

A customer (Thread-1) waits for food.

A chef (Thread-2) prepares the order and notifies the customer when food is ready.

🔹 Methods Used in Inter-Thread Communication

Method Description

wait() Makes a thread wait until notify() is called

notify() Wakes up a single waiting thread

notifyAll() Wakes up all waiting threads

8.What is Synchronization in Java?

================================

Synchronization in Java is a technique that ensures only one thread can access a shared resource at a time to prevent data inconsistency and

race conditions.

🔹 Why Use Synchronization?

To avoid multiple threads modifying shared data simultaneously.

Prevents race conditions where threads overwrite each other's changes.

Ensures data consistency in multithreading environments.

🔹 Different Ways to Achieve Synchronization

1️⃣ Synchronized Method

✔ The entire method is locked so that only one thread at a time can execute it.

🔹 Real-Time Example: ATM Withdrawal (Single User) 🏧

Imagine one person withdrawing money from an ATM at a time.

2️⃣ Synchronized Block

✔ Instead of locking the entire method, we only lock a specific block of code to improve performance.

3️⃣ Static Synchronization

✔ If a method is static, we can synchronize it so that only one thread can access it across all instances of the class.

9 What is a Deadlock Situation?

=================================

🔹 Deadlock is a situation in multithreading where two or more threads are waiting for each other to release a resource, but none can proceed,

resulting in a permanent block.

🔹 Real-Life Example of Deadlock 🚦

Imagine two cars coming from opposite directions on a narrow bridge, but neither is willing to move back. Both cars are stuck forever because each

one is waiting for the other to reverse first.

✔ Use consistent locking order, tryLock(), timeouts, and monitoring tools to prevent deadlocks.

✔ Deadlock-free applications are essential in banking, ticket booking, and online transactions.

String/oops/Array/IO STREAM/SQL/ others

1. How we can immutable class like String

==========================================

An immutable class means once an object is created, it cannot be changed. The best example is String in Java.

Steps to Create an Immutable Class

To make a class immutable, follow these rules:

Declare the class as final → Prevents subclassing.

Make all fields private and final → Prevents modification after initialization.

Do not provide setter methods → No way to change field values.

Initialize fields via a constructor → Assign values only once.

Return a copy of mutable fields → Avoid exposing internal references.

Handling Mutable Fields (Avoiding Security Issues)

If a class contains a mutable object (e.g., Date), we must return a copy instead of a direct reference.

Why Use Immutable Classes?

✅ Thread-Safety – No synchronization required.

✅ Caching & Performance – Objects can be shared safely.

✅ Security – Prevents accidental changes.

¡

3. What is hashcode and equals contract

=======================================

If two objects are equal (equals() returns true), they must have the same hashCode().

If hashCode() is the same, equals() may or may not be true.

Always override both equals() and hashCode() together to avoid data inconsistency in Hash-based collections.

4. What are the method available in object class

=================================================

Method Description

public final Class<?> getClass() Returns the runtime class of the object.

public int hashCode() Returns the hash code value for the object.

public boolean equals(Object obj) Compares this object with another object for equality.

protected Object clone()

throws CloneNotSupportedException Creates and returns a copy of this object.

public String toString() Returns a string representation of the object.

public final void notify() Wakes up a single thread that is waiting on this object's monitor.

public final void notifyAll() Wakes up all threads that are waiting on this object's monitor.

public final void wait()

throws InterruptedException Causes the current thread to wait until another thread notifies it.

public final void wait(long timeout)

throws InterruptedException Causes the current thread to wait for a specified amount of time.

public final void wait(long timeout, int nanos)

throws InterruptedException Causes the current thread to wait for a specified amount of time (in nanoseconds).

protected void finalize() throws Throwable Called by the garbage collector before an object is destroyed.

5. What is solid design principle ?

====================================

SOLID Design Principles with Real-Time Examples

The SOLID principles help in designing scalable, maintainable, and flexible software systems. Let’s explain each principle with real-world

examples that you can relate to.

1. Single Responsibility Principle (SRP)

📌 Definition: A class should have only one reason to change, meaning it should have only one responsibility.

🔹 Real-Life Example: Employee Management System

Imagine an Employee class that stores employee details, calculates salary, and generates reports.

2. Open/Closed Principle (OCP)

📌 Definition: A class should be open for extension but closed for modification.

🔹 Real-Life Example: Payment Processing System

Suppose we have a system where users can pay via Credit Card. Later, we need to add more payment methods like PayPal, UPI, etc.

3. Liskov Substitution Principle (LSP)

📌 Definition: A child class should be substitutable for its base class without altering the behavior.

🔹 Real-Life Example: Bird Behavior System

Imagine a system that models birds, where all birds should be able to fly.

4. Interface Segregation Principle (ISP)

📌 Definition: A class should not be forced to implement methods it does not use.

🔹 Real-Life Example: Printer System

Suppose we have different types of printers—some support both print & scan, while others support only printing.

5. Dependency Inversion Principle (DIP)

📌 Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions.

🔹 Real-Life Example: Database Connection System

Suppose we have a system where an Application class connects to MySQL database.

8. How to serialized a field?

==============================

To serialize an entire object, implement the Serializable interface.

If you don’t want to serialize a specific field, use the transient keyword.

9. Can we serialized static keyword

======================================

Can We Serialize a Static Field in Java?

🔹 No, static fields are NOT serialized because they belong to the class, not the object.

🔹 Serialization works on object state, and static fields are shared across all objects, so they are not part of the instance's state.

10. Differnce between JDK, JVM and JRE

**🔹 1. JDK (Java Development Kit)**

* **What it is**: A software development kit used to develop Java applications.
* **Includes**:
  + **JRE** (Java Runtime Environment)
  + **JVM** (Java Virtual Machine)
  + Development tools like javac (Java compiler), javadoc, jarsigner, etc.
* **Purpose**: For developers to **write, compile, and debug** Java programs.

📌 Example: If you're writing Java code, you need JDK.

**🔹 2. JRE (Java Runtime Environment)**

* **What it is**: A runtime environment used to **run** Java applications.
* **Includes**:
  + **JVM**
  + Core Java libraries and class files
* **Purpose**: For users who want to **run** Java applications but not develop them.

📌 Example: If you're just running a Java program (like a game or app), you only need JRE.

**🔹 3. JVM (Java Virtual Machine)**

* **What it is**: A part of JRE that executes Java bytecode (.class files).
* **Includes**:
  + Class loader
  + Bytecode verifier
  + Interpreter or JIT compiler
* **Purpose**: Converts bytecode into machine code and runs it on the host machine.

📌 Example: JVM ensures Java’s **platform independence**—write once, run anywhere.

11. How we detect memory leak in application.

=============================================

Detecting memory leaks in applications is crucial for maintaining performance and stability. Here are some common methods and tools to identify memory leaks:

Profiling Tools: Use tools like Valgrind, VisualVM, or dotMemory to monitor memory allocation and detect leaks in real-time2.

Diagnostic Tools: In environments like Visual Studio, you can use the Diagnostic Tools Window to analyze memory usage and identify leaks4.

Heap Snapshots: Take snapshots of the heap at different points in your application's lifecycle and compare them to detect unfreed memory.

Code Analysis: Review your code for common causes of leaks, such as unclosed resources, event listeners, or unmanaged memory allocations.

Logging and Monitoring: Implement logging to track memory usage patterns and identify anomalies over time.

12. How garbage collection works.(What is young generation and old generation in garbage collection

Garbage collection (GC) is a process in Java (and other languages) that automatically deallocates unused memory, ensuring efficient memory usage and preventing memory leaks.

The Java Virtual Machine (JVM) divides the heap memory (where objects are stored) into different regions to optimize the process, namely the Young Generation and Old Generation. Here's a breakdown:

Young Generation (Short-lived objects)

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

This is where new objects are allocated.

The young generation is further divided into:

Eden Space: Where objects are initially created.

Survivor Spaces: There are two survivor spaces, S0 and S1, used for intermediate stages of object promotion.

Most objects are short-lived, so they quickly become eligible for garbage collection.

The garbage collection that occurs here is called a Minor GC. It is relatively fast and happens frequently.

How it works:

Objects start in the Eden space.

If they survive a Minor GC, they are moved to a Survivor space (e.g., S0 or S1).

If they survive multiple GC cycles, they are "promoted" to the Old Generation.

Old Generation (Long-lived objects)

Stores objects that have survived several garbage collection cycles in the Young Generation.

These are typically long-lived objects like cached data or application-level data structures.

Garbage collection here is less frequent but more time-consuming. The GC process for this region is called a Major GC or Full GC.

Why is this division important?

By separating short-lived and long-lived objects, the JVM minimizes the overhead of garbage collection. Minor GCs are quick and involve only the young generation, while Major GCs (which involve the Old

Generation) are less frequent but involve more memory.

15. What is producer consumer problem.

The Producer-Consumer Problem is a classic synchronization problem in computer science, used to illustrate how processes can share resources safely in a concurrent system.

It involves two types of threads or processes:

Producer: Generates data (e.g., items) and puts it into a shared buffer.

Consumer: Fetches data from the shared buffer and processes it.

The Problem

The producer and consumer operate at different speeds, so they both need access to the shared buffer in a coordinated manner.

If the buffer is full, the producer must wait.

If the buffer is empty, the consumer must wait.

The solution must avoid issues like race conditions and deadlocks.

Real-World Analogy

Imagine a bakery:

The baker (Producer) makes cakes and places them in a display case (shared buffer).

Customers (Consumers) buy cakes from the display case.

If the display case is full, the baker has to stop baking (wait).

If the display case is empty, customers have to wait for new cakes to arrive.

**✅ Marker Interface (Short Explanation):**

A **marker interface** is an interface **with no methods**.  
It is used to **mark or tag a class** so that the **JVM or framework** can recognize and give it special behavior.

**🔹 Example:**

java

CopyEdit

class MyClass implements Serializable {

// This class can now be serialized

}

Here, Serializable is a **marker interface** — it doesn't have any methods but tells the JVM:  
👉 *"This class can be saved to a file or sent over a network."*

**✅ Real-life examples:**

* Serializable
* Cloneable
* Remote

Advance Java - Spring boot/Hibernate/Jdbc servlet jsp/

1. What is front controller

The Front Controller in Spring follows a design pattern where a single central controller handles all incoming requests and routes them to the appropriate handlers.

This is achieved through the DispatcherServlet in Spring MVC. Let me explain this concept in a diagrammatic way:

Request Flow Explanation:

Client Request: The client sends a request to the application.

Front Controller (DispatcherServlet): This central servlet intercepts the request.

Handler Mapping: The DispatcherServlet consults the HandlerMapping to determine the appropriate handler/controller for the request.

Controller: The request is passed to the controller, where the business logic is executed.

Model and View: The controller returns a Model (data) and a View (UI template name) to the DispatcherServlet.

View Resolver: The DispatcherServlet uses the ViewResolver to resolve the logical view name to an actual view (e.g., JSP, Thymeleaf).

Response: The resolved view is rendered and returned as a response to the client.

Client Request

↓

DispatcherServlet (Front Controller)

↓

Handler Mapping -- (Routes request)

↓

Controller -- (Executes business logic)

↓

Returns (Model and View)

↓

View Resolver -- (Finds appropriate view)

↓

Render View (e.g., HTML)

↓

Client Response

2. What is dependency injection difference between setter and constructor injection.

Dependency Injection (DI) means giving an object its needed resources (dependencies) from the outside instead of the object creating them itself.

This keeps the code flexible and easier to maintain.

Setter Injection: You give the object its dependencies after it's created using a public method (setter). Think of it like setting up a TV: you

buy the TV first and later plug in the cables.

Constructor Injection: You provide the object its dependencies when it's being created, through its constructor. It's like buying a TV with the cables already attached.

Key difference:

Setter injection lets you add or change dependencies later, while constructor injection ensures everything is ready upfront and cannot be changed afterward.

3. What is Actuator in SB url for actuator (name of the property)

Spring Boot Actuator is a powerful tool that provides production-ready features to help you monitor and manage your application. It offers endpoints that expose operational information about your application,

such as health checks, metrics, environment properties, and more.

Key Features of Actuator:

1)Health Monitoring: Check if your application and its components (like databases) are running properly.

2)Metrics: Gather statistics like memory usage, active threads, or request counts.

3)Environment Info: Access configuration properties or environment variables.

4)HTTP Trace: Record details about recent HTTP requests.

5)Custom Endpoints: Create your own monitoring endpoints.

How It Works:

1. Add the Actuator dependency in your pom.xml (for Maven) or build.gradle (for Gradle):

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

2. Enable or configure endpoints in your application.properties or application.yml. Example:

properties

management.endpoints.web.exposure.include=health,info

3. Access Actuator endpoints through HTTP. For example:

http://localhost:8080/actuator/health shows the application's health status.

http://localhost:8080/actuator/metrics provides metrics data.

Why Use Actuator?

>It simplifies debugging and monitoring your Spring Boot application, especially in production environments.

>Works well with monitoring tools like Prometheus or Grafana.

4. Use of devtools

=============

Spring Boot DevTools improves productivity during development by enabling features like:

Auto Restart: Automatically restarts the app when code changes.

Live Reload: Updates the browser when frontend files change.

No Template Caching: Displays template changes instantly.

Remote Debugging: Helps debug applications easily

5. for new spring boot project how we configure jpa

=====================================

1)Add Dependencies: Include spring-boot-starter-data-jpa and a database dependency (e.g., H2, MySQL) in your project.

2)Configure Database: Set database details in application.properties (like URL, username, and password).

3)Create Entity: Define your data model using the @Entity and @Id annotations.

4)Repository: Create an interface extending JpaRepository for database operations.

5)Start Using: Use the repository in a service or controller to save, fetch, or manage data.

6. What are the starter in Spring boot

Common Spring Boot Starters:

1)spring-boot-starter-web: For building web applications, including REST APIs.

2)spring-boot-starter-data-jpa: For working with JPA and relational databases.

3)spring-boot-starter-test: For testing, including JUnit and Mockito libraries.

4)spring-boot-starter-security: For adding authentication and security to your app.

5)spring-boot-starter-thymeleaf: For Thymeleaf-based template engines.

6)spring-boot-starter-logging: For logging with Logback (default) or Log4j2.

7. How we can read the property from properties file in spring boot

To read properties in Spring Boot:

Using @Value: Inject specific property values into fields.

@Value("${app.name}")

private String appName;

Using @ConfigurationProperties: Map multiple related properties to a POJO.

@ConfigurationProperties(prefix = "app")

private String name;

Using Environment: Fetch properties dynamically.

environment.getProperty("app.name");

These methods let you access values from application.properties!

8. Difference between Authentication and Authorization

Authentication:

>It's about verifying who a user is.

>Ensures that the user is genuine.

Example: Logging in with a username and password.

Authorization:

>It's about verifying what a user is allowed to do.

>Grants access to specific resources or actions.

Example: A logged-in user may have access to files, but not admin settings.

In short:

Authentication = "Who are you?"

Authorization = "What can you do?"

9. what is the scope of Spring bean

1. Singleton (Default Scope)

Description: A single instance of the bean is created and shared across the entire application context.

Use Case: Stateless beans that are shared across multiple components.

2. Prototype

Description: A new instance of the bean is created each time it is requested.

Use Case: Stateful beans or beans requiring unique instances per request.

3. Request (Web Application Scope)

Description: A new bean instance is created for each HTTP request.

Use Case: Beans specific to an HTTP request in web applications.

4. Session (Web Application Scope)

Description: A single instance of the bean is created for each HTTP session.

Use Case: Beans tied to a specific user session.

5. Application (Web Application Scope)

Description: A single instance of the bean is shared across the entire application (similar to a ServletContext).

Use Case: Beans specific to the application lifecycle in web applications.

6. WebSocket (Web Application Scope)

Description: A single instance is created and tied to a WebSocket session.

Use Case: Beans specific to a WebSocket session.

10. use @Qualifier annotation

The @Qualifier annotation in Spring is used to resolve the issue of ambiguity when you have multiple beans of the same type and need to specify which one to

inject into a particular dependency. This is especially helpful in scenarios where Spring's automatic wiring (@Autowired) cannot determine which bean to choose.

Step 1: Define Multiple Beans of the Same Type

import org.springframework.stereotype.Component;

@Component("serviceOne")

public class ServiceOne implements MyService {

@Override

public String serve() {

return "Service One";

}

}

@Component("serviceTwo")

public class ServiceTwo implements MyService {

@Override

public String serve() {

return "Service Two";

}

}

Step 2: Use @Autowired with @Qualifier to Resolve the Ambiguity

@Component

public class Client {

private final MyService myService;

@Autowired

public Client(@Qualifier("serviceOne") MyService myService) {

this.myService = myService;

}

public void doSomething() {

System.out.println(myService.serve());

}

}

11. what database using and how to connect Spring boot application

I have used my sql workbench database, first i need to add database dependency in pom.xml

after that i need to add database properties in application.properties file like username, password, url, driver class name, etc

12. What are module in Spring framework

1)Core Container: For managing beans and dependency injection.

2)Data Access: Handles database operations (e.g., JDBC, ORM integration).

3)Web: Supports web and MVC-based applications.

4)AOP: Adds features like logging and transactions using aspects.

5)Security: Manages authentication and authorization.

6)Testing: Tools for unit and integration testing.

7)Others: Modules like Batch (for large-scale processing) and WebFlux (for reactive apps).

13. how to write api to save employee object

============================================

14. what cohesion(loose coupled) and coupling(tightly coupled)

1. Cohesion (Single Responsibility)

Cohesion means that a class or a part of your program should focus on one specific task or responsibility.

>Real-life Example: Think of a calculator. It has buttons to perform calculations, like addition or subtraction. The calculator:

2. Coupling (Dependency Between Classes)

>Coupling refers to how much one class depends on another. Lower coupling makes it easier to change or reuse parts of the program.

>Real-life Example: Think of a TV remote. A remote is loosely coupled with the TV—it can work with multiple TV brands. If you replace the TV,

the same remote can still work. This is loose coupling.

>On the other hand, if a remote only worked with one specific TV model, that's tight coupling.

15. Benefit of using jpa over Hibernate.

1)Standardization: JPA is vendor-neutral, so you can switch from Hibernate to another provider (like EclipseLink) without major code changes.

2)Portability: Your application remains adaptable to various persistence providers.

3)Cleaner Code: JPA's annotations and interfaces (like @Entity, EntityManager) are simple and less tied to a specific framework.

4)Better Integration: JPA works seamlessly with frameworks like Spring Boot and Spring Data JPA.

5)Flexibility: You can still access Hibernate-specific features when needed.

16. what are the stages of Hibernate ( 3 state of hiberante entity)

1)Transient: The object is created but not saved in the database and not associated with any Hibernate session. Example: Employee emp = new Employee();

2)Persistent: The object is saved in the database and is managed by a Hibernate session. Example: session.save(emp);

3)Detached: The object was saved before, but the session is now closed. Changes won't be synchronized with the database until reattached.

Example: session.close(); emp.setName("Updated");

20. How to avoid status 500

============================

To avoid 500 Internal Server Errors, follow these steps:

1)Log Errors: Use proper logging to identify issues.

2)Handle Exceptions: Implement @ControllerAdvice and @ExceptionHandler for global error handling.

3)Validate Input: Ensure user input is correct using validation annotations like @NotNull.

4)Optimize Queries: Avoid complex or inefficient database queries.

5)Monitor Resources: Check CPU, memory, and server load.

6)Set Timeouts: Prevent long-running tasks or infinite loops.

7)Test Thoroughly: Test edge cases and simulate high loads.

21. How we can get the data from database in Hibernate.

22. Difference between get and load method in Hibernate.

1)get(): Fetches the data immediately from the database. If the record doesn't exist, it returns null.

2)load(): Fetches data lazily by creating a proxy object. The database is accessed only when the data is needed. If the record doesn't exist, it throws an ObjectNotFoundException.

23. What is serial version UID

The serialVersionUID is a unique identifier used in Java serialization. It helps ensure that during the deserialization process,

the sender and receiver of a serialized object are using compatible versions of the class.

Key Points About serialVersionUID:

1)Purpose: It prevents InvalidClassException by verifying that the serialized object's class is compatible with the current version of the class.

2)Default Behavior: If you don’t explicitly declare a serialVersionUID, Java generates one automatically based on the class structure.

3)Customization: It's recommended to explicitly define it, as changes in the class structure might lead to incompatibility with the auto-generated ID.

public class Employee implements Serializable {

private static final long serialVersionUID = 1L; // Explicitly declared

private String name;

private int age;

// Getters and Setters

}

serialVersionUID = 1L ensures the compatibility of this class across different versions.

24. What is solid principle

1)S: Single Responsibility Principle (SRP) A class should have only one reason to change, meaning it should have only one responsibility or functionality.

This ensures better modularity and reduces the impact of changes.

2)O: Open/Closed Principle (OCP) Software entities like classes, modules, and functions should be open for extension but closed for modification.

In essence, you can add new functionality without altering the existing code, which helps prevent introducing bugs.

3)L: Liskov Substitution Principle (LSP) Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

This ensures that derived classes adhere to the behavior expected by their base classes.

4)I: Interface Segregation Principle (ISP) A class should not be forced to implement interfaces it does not use. It is better to have smaller,

more specific interfaces rather than a large, all-encompassing one.

5)D: Dependency Inversion Principle (DIP) High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should

not depend on details, but details should depend on abstractions. This principle promotes flexibility and reduces coupling between components.

25. How we can connect multiple database.(Profile concept)

To connect multiple databases using the profile concept, you define separate configurations for each database and activate the desired one through a profile.

Here's a concise breakdown:

1)Create Configuration Files:

Example: application-primary.properties for one database and application-secondary.properties for another.

2)Set Profiles:

Activate the desired database by setting spring.profiles.active=primary or secondary.

3)Define Beans for DataSources:

Use @Bean and @ConfigurationProperties for each database, like primaryDataSource and secondaryDataSource.

4)Switch Profiles Dynamically:

Annotate configurations with @Profile("primary") or @Profile("secondary").

5)Specify Database Use in Services:

Use @Qualifier to choose which database (primary or secondary) your service should access.

27. What is the @SpringBootApplication annotation.

The @SpringBootApplication annotation in Spring Boot is a convenience annotation that combines three crucial annotations to simplify application configuration.

It is typically placed on the main class of a Spring Boot application. Here's what it does:

1)@EnableAutoConfiguration: Automatically configures Spring application based on the dependencies you include in the project, reducing the need for extensive manual configurations.

2)@ComponentScan: Scans the package where the class is located (and its sub-packages) to detect Spring components (like @Controller, @Service, etc.) and register them in the Spring context.

3)@Configuration: Indicates that the class is a source of bean definitions for the Spring application context.

29. What is spring initilizer

The Spring Initializer (or start.spring.io) is an online tool provided by Spring to quickly bootstrap a new Spring Boot project.

It simplifies the process of setting up a project by generating all the essential files and dependencies for you. Here's how it works:

1.Choose Project Settings: You can select:

>The build tool: Maven or Gradle.

>The programming language: Java, Kotlin, or Groovy.

>The Spring Boot version.

2.Add Dependencies: Select the libraries or frameworks you need for your project, such as Spring Web, Spring Data JPA, Spring Security, etc.

3.Generate Project: Once the configuration is done, click "Generate" to download a pre-configured project as a ZIP file. This project includes:

pom.xml (for Maven) or build.gradle (for Gradle).

A basic folder structure with src/main for code and src/test for test files.

An application.properties file for configurations.

Import into Your IDE: Extract the ZIP, import it into your favorite IDE (like IntelliJ or Eclipse), and start coding immediately.

30. How we can change the tomcat server in spring boot

To change the default Tomcat server in Spring Boot:

1)Exclude Tomcat: Remove the Tomcat dependency from your project configuration (e.g., in pom.xml or build.gradle).

2)Add New Server: Include the dependency for the desired server, like Jetty or Undertow.

3)Verify: Run your application, and you'll see the new server (e.g., Jetty) in the logs.

31. How we can disable any configuration.( use @exclude annotation)

To disable a specific configuration in Spring Boot, you can use the @SpringBootApplication's exclude attribute, which allows you to prevent certain auto-configuration

classes from being loaded. Here's how to do it:

Example:

Suppose you want to disable DataSourceAutoConfiguration. You can do it like this:

@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

Short Explanation:

The exclude parameter in @SpringBootApplication stops specific auto-configurations from being loaded.

Use it when you want more control over which configurations are applied to your application.

32. What are the best practice while writing sql queries.

===========================================================

1)Use only the required columns (SELECT column1, column2) instead of SELECT \*.

2)Apply proper indexes on frequently searched columns.

3)Write clear and readable queries with proper formatting.

4)Use JOIN instead of subqueries when possible.

5)Add efficient WHERE clauses and avoid functions on indexed columns.

6)Limit results with LIMIT or pagination.

7)Use parameters to avoid hardcoding and prevent SQL injection.

8)Analyze and optimize query execution plans using tools like EXPLAIN.

Microservices

Components of Microservices Architecture

**🧩 1. Service Components**

Each **microservice** is:

* A small, focused, and independent service
* Handles **a single business function** (like User, Order, Payment)
* Has its own **codebase**, **database**, and logic

**🔗 2. API Gateway**

* Acts as the **entry point** to the system
* Routes requests to appropriate microservices
* Handles **authentication**, **logging**, **rate limiting**, etc.

📌 Tools: **Spring Cloud Gateway**, **Zuul**, **Kong**

**🔄 3. Service Discovery**

* Allows services to **find each other** dynamically
* Especially useful when services are scaled or move to new locations

📌 Tools: **Eureka**, **Consul**, **Zookeeper**

**🗃️ 4. Database per Service**

* Each microservice has its **own database**
* Ensures **loose coupling** and **data isolation**

📌 Example:

* User Service → user\_db
* Order Service → order\_db

**📨 5. Inter-Service Communication**

* Services communicate using **REST APIs**, **gRPC**, or **messaging queues**

📌 Protocols: HTTP/JSON, gRPC  
📌 Tools: **RabbitMQ**, **Kafka**

**🛠️ 6. Configuration Management**

* Centralized way to manage config files for each service

📌 Tools: **Spring Cloud Config**, **Consul**, **Vault**

**📈 7. Monitoring & Logging**

* Track service health, logs, and performance

📌 Tools:

* **ELK Stack** (Elasticsearch, Logstash, Kibana)
* **Prometheus + Grafana**
* **Zipkin**, **Jaeger** for tracing

**🔒 8. Security**

* Ensures **authentication & authorization**
* Often handled at the **API Gateway**

📌 Tools: **OAuth2**, **JWT**, **Keycloak**

**📦 9. Containerization & Orchestration**

* Package services using **Docker**
* Manage and scale them using **Kubernetes**

1. How to Microservices communicate with each other

====================================================

Microservices communicate with each other using various methods depending on the architecture and use case. Here are the key ways:

1. Synchronous Communication

Services directly call each other in real-time, waiting for a response.

Common protocols: HTTP/REST, gRPC.

Example: Service A sends an HTTP request to Service B and waits for the response.

2. Asynchronous Communication

Services communicate without waiting for an immediate response, ensuring loose coupling.

Common methods: Message brokers like RabbitMQ, Kafka, or ActiveMQ.

Example: Service A publishes a message to a queue, and Service B processes it later.

3. Event-Driven Communication

Services react to events instead of direct requests.

Event producers (services) generate events, and consumers (other services) subscribe to them.

Example: A user registration event triggers another service to send a welcome email.

4. Service Discovery

Microservices use a service discovery tool (e.g., Eureka, Consul) to dynamically find the location of other services.

This avoids hardcoding of service addresses.

5. API Gateway

An API Gateway acts as a single entry point for client requests, handling routing to multiple services.

Example: A gateway routes a client request to the appropriate microservice behind the scenes.

2. What is synchronous and asynchronous way of communication in Microservices

1. Synchronous Communication

Definition: One service sends a request to another and waits for an immediate response before continuing.

Example: Service A calls Service B via an HTTP REST API, waits for the result, and then proceeds.

Common Protocols: HTTP/REST, gRPC.

Use Case: Best for tasks requiring real-time responses, like login validation or order processing.

2. Asynchronous Communication

Definition: One service sends a message to another but doesn’t wait for a response. Instead, it proceeds with its work, and the response (if any) comes later.

Example: Service A sends a message to a message broker (e.g., RabbitMQ, Kafka), and Service B processes it when it can.

Common Tools: Message queues, event streams, Pub/Sub systems.

Use Case: Suitable for tasks that don’t need immediate feedback, like sending notifications or processing large batches.

Key Difference:

Synchronous: Real-time, tightly coupled, blocks until the response is received.

Asynchronous: Decoupled, allows better scalability, doesn’t block.

3. Component of Microservices artictecture

The components of a microservices architecture ensure the seamless functioning of distributed services. Here are the key components:

1. API Gateway

Acts as a single entry point for client requests.

Handles routing, load balancing, authentication, and rate limiting.

2. Service Registry

Maintains a directory of all available microservices and their locations (e.g., Eureka, Consul).

Allows services to discover each other dynamically.

3. Service Communication

Synchronous: HTTP/REST, gRPC for real-time interaction.

Asynchronous: Message queues (RabbitMQ, Kafka) for event-driven or delayed processing.

4. Database for Each Service

Each microservice manages its own database (Database Per Service pattern) to ensure loose coupling.

5. Message Broker

Facilitates asynchronous communication using Pub/Sub mechanisms (e.g., RabbitMQ, Kafka, ActiveMQ).

6. Load Balancer

Distributes requests evenly across instances of a microservice to handle high traffic and ensure reliability.

7. Distributed Logging and Monitoring

Tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Prometheus-Grafana for tracking and troubleshooting.

8. Configuration Management

Centralized management of service configurations using tools like Spring Cloud Config or Consul.

9. Service Deployment

Tools like Docker and Kubernetes are used for containerization, orchestration, and scaling.

10. Security Components

Implement authentication (e.g., OAuth2, JWT) and secure service-to-service communication (e.g., TLS/SSL).

4. What is circuit breaker

A circuit breaker in microservices is a design pattern used to handle failures gracefully and improve system resilience.

It prevents a service from repeatedly trying to call another service that is experiencing issues, which could lead to cascading failures.

How it Works:

Closed State: All requests are allowed to pass to the dependent service. If failures occur, they are recorded.

Open State: After a threshold of failures is reached, the circuit "opens," and further requests are blocked temporarily to avoid overwhelming the failing service.

Half-Open State: The system periodically allows some requests to check if the dependent service has recovered. If successful, the circuit closes; otherwise, it reopens.

Real-Life Analogy:

Think of a household circuit breaker. When there's too much load or a fault, it "trips" to prevent damage. Similarly, in microservices, the circuit breaker stops calls

to a failing service to avoid further issues.

Implementation Example:

Libraries like Hystrix (Netflix) or Resilience4j are commonly used to implement the circuit breaker pattern in Java-based microservices.

This pattern is crucial for building reliable systems.

5. Demerit/ disadvantage/challenges of Microservices

While microservices have significant advantages, they also come with several challenges and disadvantages:

1. Increased Complexity

Managing multiple small services is more complex than handling a monolithic application.

Requires tools for service discovery, monitoring, and communication.

2. Higher Resource Consumption

Each microservice may need its own runtime environment, leading to increased memory and CPU usage.

3. Difficult Debugging and Testing

Debugging distributed systems is harder because issues may span across multiple services.

Integration testing requires coordination among services.

4. Network Latency

Communication between services happens over the network, introducing latency and possible failure points.

5. Data Consistency

Managing consistency across distributed databases can be challenging, requiring eventual consistency or complex transaction management (e.g., Saga Pattern).

6. Deployment Overhead

Frequent deployments of multiple services can require robust CI/CD pipelines and containerization tools like Docker and Kubernetes.

6. Database design pattern use in Microservices

In microservices, different database design patterns are used to manage data efficiently while maintaining service independence. Here are the commonly used database patterns:

1. Database per Service

Each microservice has its own dedicated database.

Ensures loose coupling and independent scaling of services.

Challenge: Data consistency across services.

2. Shared Database

Multiple microservices share a single database.

Simplifies consistency but creates tight coupling, reducing scalability.

3. Saga Pattern

Ensures data consistency in distributed transactions using a series of compensating actions.

Used for long-running processes that span multiple services.

4. Event Sourcing

Captures all changes to data as a sequence of events, rather than storing just the current state.

Helps in auditing and rebuilding application state.

5. CQRS (Command Query Responsibility Segregation)

Splits the database into separate models for write operations (commands) and read operations (queries).

Improves performance and scalability by optimizing each use case.

6. API Composition

Instead of directly querying databases, services expose APIs to provide the required data.

Reduces direct database dependencies but adds communication overhead.

7. Change Data Capture (CDC)

Tracks changes in one database and propagates them to others through events.

Useful for ensuring synchronization without direct coupling.

7. How we do login using Microservices

Implementing login in a microservices architecture involves several components working together to authenticate and manage users securely. Here's an outline:

1. Authentication Service

Create a dedicated microservice for authentication (e.g., Auth Service) to handle login and token generation.

When a user logs in, this service verifies the credentials and generates a JWT (JSON Web Token) or an OAuth2 token.

2. User Database

Store user credentials securely (e.g., hashed passwords) in a database managed by the Auth Service.

3. Client Interaction

The client (e.g., web app, mobile app) sends the user's credentials (e.g., username and password) to the Auth Service via an API call.

4. Token-Based Authentication

On successful login, the Auth Service generates a token (e.g., JWT) and sends it back to the client.

5. Token Validation

For subsequent requests to other microservices, the client includes the token in the Authorization header.

Each microservice validates the token before processing the request, ensuring the user is authenticated.

6. Service Communication

Microservices can validate tokens using:

A shared secret for JWT.

A central token verification service.

A public key for verifying signed tokens.

7. API Gateway Integration

An API Gateway can handle authentication and route user requests to the appropriate microservice.

It can also validate tokens and block unauthorized requests, simplifying security for individual services.

The token contains user information and permissions and is signed for security.

8. What are the security technique in Microservices

1. Authentication and Authorization

Use token-based authentication mechanisms like OAuth2 or JWT to authenticate users and services.

Apply Role-Based Access Control (RBAC) for fine-grained authorization.

2. API Gateway Security

Use an API Gateway to handle authentication, rate-limiting, and request validation before routing requests to microservices.

Enforce HTTPS for secure data transmission.

3. Service-to-Service Authentication

Use mutual TLS (mTLS) to secure communication between services.

Implement service identity using tools like Istio or Consul.

4. Data Encryption

Encrypt sensitive data both at rest (using database encryption) and in transit (using protocols like TLS/SSL).

5. Secure Configuration Management

Store sensitive configuration details (e.g., API keys, credentials) securely using tools like HashiCorp Vault or AWS Secrets Manager.

6. Network Segmentation

Use container orchestration tools (e.g., Kubernetes) to isolate microservices in private networks or namespaces.

7. Input Validation

Validate and sanitize user inputs to prevent injection attacks like SQL injection or cross-site scripting (XSS).

8. Rate Limiting and Throttling

Prevent abuse or denial-of-service attacks by limiting the number of requests a user or service can make.

9. Logging and Monitoring

Use centralized logging and monitoring tools (e.g., ELK Stack, Prometheus) to detect and respond to suspicious activities.

10. Secure APIs

Follow REST API security best practices, like avoiding sensitive data in URLs and using strong authentication headers.

9. How we can manage transaction in microservices.

Managing transactions in microservices can be challenging due to their distributed nature. Here are common approaches used to ensure consistency and reliability:

1. Two-Phase Commit (2PC)

A distributed transaction protocol where a coordinator ensures all services commit or roll back the transaction together.

Drawback: High latency and tightly coupled services, so it’s rarely used in microservices.

2. Saga Pattern

Choreography: Each service performs its operation and publishes an event for the next service in the workflow. Services react to these events.

Orchestration: A central orchestrator coordinates the transaction by invoking each service in sequence and handling failures with compensating actions.

Example: If a payment fails, the compensating action cancels a previously confirmed order.

3. Eventual Consistency

Instead of strict consistency, services aim for data to eventually synchronize across systems.

Implement with message queues (e.g., RabbitMQ, Kafka) to ensure asynchronous communication.

4. Outbox Pattern

A service writes to its database and an "outbox" table as part of the same transaction.

Another process reads the outbox table and publishes messages to other services, ensuring no messages are lost.

5. Distributed Locking

Use distributed locks to manage resources shared across multiple services (e.g., using Redis or ZooKeeper).

Prevents concurrent updates to the same resource.

10. Suppose there is a call from microservices A to B but is down so what will happen in that scenario.

If microservice B is down and A makes a call to it, here’s what typically happens depending on the setup:

1. Without Resilience Mechanisms:

Failure: The call from A to B will fail, resulting in errors like Connection Timeout or Service Unavailable.

Impact: It could lead to cascading failures if A depends heavily on B.

2. With Resilience Mechanisms:

Resilience strategies can handle such failures gracefully:

Circuit Breaker: If B is repeatedly unavailable, the circuit breaker will "open," preventing further calls to B for a defined period. This avoids overwhelming the system.

Fallback Logic: A fallback mechanism can provide default behavior or cached data instead of relying on B.

Retry Logic: Service A may retry the call to B a certain number of times before giving up.

Timeouts: Configured timeouts ensure A doesn’t wait indefinitely for B to respond.

Queue-Based Communication: If the call is asynchronous (e.g., using Kafka), the request can be queued and processed once B is back online.

In Short:

If B is down:

Without resilience, A will fail.

With resilience mechanisms like circuit breakers, fallbacks, retries, or queues, the system can handle the failure more gracefully.

11. What is used of API Gateway.

An API Gateway acts as a single entry point for client requests in a microservices architecture. It is a critical component that simplifies the interaction between clients and the microservices by providing

several essential functionalities:

1)Routing Requests: It routes client requests to the appropriate microservices based on the API path.

2)Load Balancing: Distributes incoming requests across multiple instances of services to ensure scalability and reliability.

3)Authentication and Authorization: It handles security mechanisms like verifying API keys, tokens, or enforcing role-based access controls.

4)Protocol Translation: Converts between protocols (e.g., HTTP to gRPC) to make the communication seamless for clients.

5)Rate Limiting and Throttling: Limits the number of requests from clients to prevent overuse and ensure fair usage.

6)Caching: Stores frequently accessed responses temporarily to reduce load on backend services.

7)Centralized Logging and Monitoring: Collects and provides insights into API usage, errors, and performance metrics.

In essence, it simplifies client communication, reduces complexity in microservices, and centralizes cross-cutting concerns. If you're exploring API Gateways for your projects,

tools like Kong, Amazon API Gateway, or Spring Cloud Gateway might pique your interest!

Design Pattern

1. What are design pattern ?

2. Which scenario we use factory design pattern ?

The Factory Design Pattern is a creational design pattern used to create objects without specifying the exact class of the object that will be created.

Instead of directly instantiating objects, you use a factory method to encapsulate the object creation logic. This pattern is particularly useful when the exact type of object to be created depends on conditions

at runtime.

Real-Life Analogy:

Think of an ice cream factory. Based on the flavor you request (vanilla, chocolate, or strawberry), the factory produces the appropriate type of ice cream.

The customer (you) doesn’t worry about the process of making the ice cream; you just receive the correct one.

In programming, the factory takes the "flavor" (input) and decides which "ice cream type" (object) to create.

Benefits of the Factory Pattern:

1)\_Decouples Object Creation: The client code does not need to know the exact class of the object.

2) Scalability: Adding a new type of object (like Triangle) just requires creating a new class and updating the factory, without changing the client code.

3) Reusability: Centralizes the object creation logic, making it reusable across the application.

Real-World Uses:

1)Database Connections: A factory can create connections to different databases (e.g., MySQL, PostgreSQL) based on configuration.

2)Document Generators: A factory might create PDF or Word documents depending on the required format.

3)Notification Systems: A factory can create different types of notifications (e.g., SMS, Email, Push) based on user preferences.

3. What is builder design pattern

The Builder Design Pattern is a creational design pattern that helps construct complex objects step by step. It allows for creating different types and representations

of an object using the same construction code. This pattern is particularly useful when an object has multiple optional attributes, as it promotes readability and flexibility in the code.

Real-Life Analogy:

Think of building a burger at a fast-food restaurant. A basic burger has bread, but you can customize it by adding cheese, lettuce, tomatoes, sauces, etc., depending on your

preferences. The burger is built step by step, and at the end, you get a complete, customized burger.

In programming, the Builder Pattern works similarly—it builds objects in a stepwise manner and allows customization.

Example in Java:

Let’s say you’re creating a House object, which has optional features like swimming pool, garden, and garage. Instead of creating multiple constructors, you can use a

Builder Pattern:

Benefits of the Builder Pattern:

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

Simplifies Object Creation: Avoids a large number of constructors for different combinations of parameters.

Readability: Easy to understand the steps used to construct the object.

Immutability: The final object is immutable as all attributes are set during construction.

Real-World Uses:

Creating objects like HTTP requests, where you can add headers, body, etc.

Document generation, where the content and formatting are built step by step.

UI building, where you can add components like buttons, labels, and menus.

4. What is singleton design pattern

Singleton Design Pattern for you with a real-life example!

Imagine you're in a house with only one electricity meter. Regardless of how many appliances or rooms you have, they all share the same meter to measure power usage.

You wouldn’t install a separate meter for each room—that would be inefficient and confusing. Instead, you have one electricity meter that acts as a shared resource for the whole house.

Now, in programming terms:

1)The electricity meter is the Singleton—it’s a shared resource that is used everywhere in your application.

2)The Singleton pattern ensures that there is always one and only one instance of the electricity meter in your house (or object in your code).

Here’s a simple example in Java:

In real-world coding, you might use a Singleton for things like:

1)Logging: A single logger instance ensures that all parts of your application log messages in the same way.

2)Configuration: A central object manages settings like database URLs, API keys, or file paths.

3)Connection Pooling: A shared instance manages multiple database connections efficiently.

My Interview Question.

======================

1) What algorithm use in spring security ?

Spring Security itself is a framework, not a single algorithm — it uses multiple algorithms, depending on what feature you're using (authentication,

authorization, password encoding, token generation, etc.).

Here’s a breakdown of main algorithms used inside Spring Security:

🔒 1. Password Encoding

When you store passwords securely, Spring Security uses password hashing algorithms like:

BCrypt (default and recommended)

PBKDF2 (Password-Based Key Derivation Function 2)

SCrypt

Argon2

Default: Since Spring Security 5, BCryptPasswordEncoder is the default.

✅ BCrypt is good because it is slow (by design) and uses a random salt every time.

🛡️ 2. Authentication Mechanisms

Depending on the authentication type, different algorithms are used:

Authentication Type Algorithm / Concept

Basic Authentication Base64 encoding (of username:password)

Form Login Depends — usually uses BCrypt for password verification

JWT Authentication HMAC SHA-256 (if using HS256 signing) or RSA/ECDSA if using public/private key signing

📜 3. Authorization

No special "algorithm" — it uses AccessDecisionManager with rules like role checks, authority checks, etc.

Supports expressions like hasRole('ADMIN'), hasAuthority('READ\_PRIVILEGES').

🔑 4. Token Based Authentication (JWT, OAuth2)

If you are doing OAuth2 or JWT authentication:

JWT tokens: Signed using HMAC SHA-256 (symmetric) or RSA/ECDSA (asymmetric)

OAuth2 uses different flows but again relies on signing algorithms.

🧠 5. Session Management

Spring Security can manage sessions (store security context), but no encryption algorithm is used for the session itself unless you configure

special session security.

2)what is spring batch ?

========================

Spring Batch is a framework for batch processing in Java.

It helps you read large amounts of data, process it, and then write it somewhere — in chunks or step-by-step — automatically handling retries,

skipping errors, and logging.

Batch processing means:

Running big jobs like reading 10 lakh (1 million) records from a file or database

Doing some calculation/validation

Saving the result into a database, file, etc.

🚀 In short:

Feature Meaning

Read Take data from database, CSV, XML, etc.

Process Transform, validate, or filter the data

Write Save into database, another file, etc.

Manage Retry failed records, skip errors, create logs

🛠 Example:

Read 1 lakh customer records from a CSV file

Update their account balances

Write the updated records back to the database

🔥 Important Components:

Job → the full batch job

Step → a piece of a job (read → process → write)

ItemReader → reads data

ItemProcessor → processes data

ItemWriter → writes data

🧠 One-line definition:

"Spring Batch is used to efficiently handle large data processing tasks in a structured, reliable, and scalable way."

3)what is CORS policy ?

=======================

CORS stands for Cross-Origin Resource Sharing.

It is a security rule that browsers follow to allow or block requests between different domains (origins).

🔥 In simple words:

If your frontend (like React, Angular) is running at http://localhost:3000

And your backend (Spring Boot API) is running at http://localhost:8080

Then CORS decides whether your frontend is allowed to call the backend.

If CORS is not allowed, the browser blocks the call and shows an error like:

"CORS policy: No 'Access-Control-Allow-Origin' header is present."

🛡 Why CORS Exists?

👉 To protect users from malicious websites trying to send fake requests to your server.

🚀 Quick Example:

Situation CORS Needed?

Frontend and backend on same domain ❌ No

Frontend and backend on different domains or ports ✅ Yes

🧠 One-line definition:

"CORS policy controls whether a browser can call an API from a different domain."

4)what is the capacity of ArrayList ?

======================================

Capacity = how many elements the ArrayList can hold internally before it needs to resize (grow bigger).

Size = how many elements are actually stored right now.

Default capacity ==> 10 (if you create a new ArrayList without giving size)

Capacity increases ==> Automatically when ArrayList is full

Growth formula ==>When full, new capacity = old capacity + (old capacity / 2) → roughly 1.5x

8)what is auto-closable interface?

AutoCloseable is a Java interface that has one method:

void close() throws Exception;

✅ It is used to automatically close resources (like files, database connections) after use.

try (FileInputStream fis = new FileInputStream("test.txt")) {

// read file

}

// fis.close() is automatically called after try block

FileInputStream implements AutoCloseable.

No need to manually call fis.close() — Java does it for you!

🧠 One-line definition:

AutoCloseable is an interface that allows Java to automatically close resources after you are done using them."

AutoCloseable introduced in Java 7 (for try-with-resources).

Many Java classes like FileInputStream, BufferedReader, Connection implement it.

9)What are the method in String class ?

=======================================

Category Methods

Length ==> length()

Character Access ==> charAt(int index)

Comparison ==> equals(), equalsIgnoreCase(), compareTo(), compareToIgnoreCase()

Searching ==> indexOf(), lastIndexOf(), contains(), startsWith(), endsWith()

Substring ==> substring(int beginIndex), substring(int beginIndex, int endIndex)

Case Conversion ==> toLowerCase(), toUpperCase()

Trim/Remove ==>trim(), strip(), stripLeading(), stripTrailing()

Replace ==> replace(), replaceAll(), replaceFirst()

Split/Join ==> split(), join()

Conversion ==> toCharArray(), valueOf(), getBytes()

Empty/Blank Check ==> isEmpty(), isBlank()

Formatting ==> format()

Interning ==> intern()

Matches (Regex) ==> matches()

11)If string is "mahesh" if i want to replace h with any other character how we can do that?

==============================================================================================

String str = "mahesh";

String newStr = str.replace('h', 'z');

System.out.println(newStr); // Output: "mazezs"

12)Difference between JPQL and native query ?

13)Explain spring MVC cycle?

============================

1️⃣ Client (Browser) sends a Request

Example: User hits http://localhost:8080/getEmployees

2️⃣ Front Controller (DispatcherServlet) receives the request

DispatcherServlet is the central point.

It checks where to send the request.

3️⃣ Handler Mapping decides the correct Controller

Spring checks which Controller method should handle the request.

(Based on URL mapping like @RequestMapping or @GetMapping).

4️⃣ Controller processes the request

Controller handles the logic.

May call the Service layer or Database.

Then returns a ModelAndView or Response.

5️⃣ View Resolver decides which View to show

View Resolver takes the logical view name (like "employeeList") and maps it to a physical view (like a JSP page).

6️⃣ Response is sent back to Browser

DispatcherServlet sends the final HTML/JSON response back to the Client.

14)How substring method work ?

===============================

String str = "mahesh";

// Only start index

String part1 = str.substring(2);

System.out.println(part1); // Output: "hesh" (starts from index 2 till end)

// Start and end index

String part2 = str.substring(1, 4);

System.out.println(part2); // Output: "ahe" (index 1 to index 3)

15)How we can handle custom exception in java ?

===============================================

1️⃣ Create your own Exception Class

Extend Exception (for checked exception)

or RuntimeException (for unchecked exception).

// Custom Checked Exception

class MyCustomException extends Exception {

public MyCustomException(String message) {

super(message);

}

}

2️⃣ Throw the Custom Exception

You can throw it manually where needed.

public void validateAge(int age) throws MyCustomException {

if (age < 18) {

throw new MyCustomException("Age must be 18 or above!");

}

}

3️⃣ Handle it using try-catch

You can catch the custom exception.

public static void main(String[] args) {

try {

new Main().validateAge(15);

} catch (MyCustomException e) {

System.out.println("Custom Exception caught: " + e.getMessage());

}

}

16)What is advantages and disadvantages of Multhithreading ?

=============================================================

Advantages of Multithreading:

Advantage Description

1. Faster execution Multiple threads run at the same time → better performance.

2. Better CPU usage CPU stays busy, not idle → efficient resource usage.

3. Easy multitasking You can do many tasks at once (like downloading + playing music).

4. Improved application responsiveness UI applications (like games, websites) don't freeze while background work happens.

5. Better resource sharing Threads share memory easily (not like processes which are heavy).

❌ Disadvantages of Multithreading:

Disadvantage Description

1. Complex to program Code becomes harder to write and debug.

2. Risk of thread interference Threads may corrupt data if not handled properly (example: two threads changing same variable).

3. Risk of deadlocks Threads can block each other forever if locks are not handled well.

4. Harder to maintain Multithreaded applications are difficult to test and fix bugs.

5. Overhead Too many threads can cause context switching overhead → slower instead of faster.

17)How deployment happen in microservices ?

===========================================

How Deployment Happens in Microservices:

In Microservices, each service is built, packaged, and deployed independently.

🛠 Step-by-Step Deployment Flow:

1️⃣ Build Each Microservice

Each microservice is developed separately.

Example: account-service, payment-service, notification-service, etc.

2️⃣ Package into Executables

Each service is packaged (like a .jar, .war, or Docker image).

✅ Example:

Maven/Gradle → builds .jar

Docker → creates an image

3️⃣ Push to Repository

Built artifacts are stored in repositories.

JARs → Nexus, Artifactory.

Docker Images → DockerHub, AWS ECR, GitHub Container Registry.

4️⃣ Deploy to Environment

Services are deployed to environments like:

Development (DEV)

Testing (QA/UAT)

Production (PROD)

5️⃣ Use Orchestration Tools (optional)

Tools like Kubernetes, Docker Swarm, or AWS ECS manage deployments automatically.

They can scale up/down, restart services, balance load, etc.

📦 Deployment Methods:

Deployment Type Meaning

Docker Containers Run each service inside its own lightweight container.

Virtual Machines Each service on a separate VM.

Cloud Platforms AWS, Azure, GCP manage deployment.

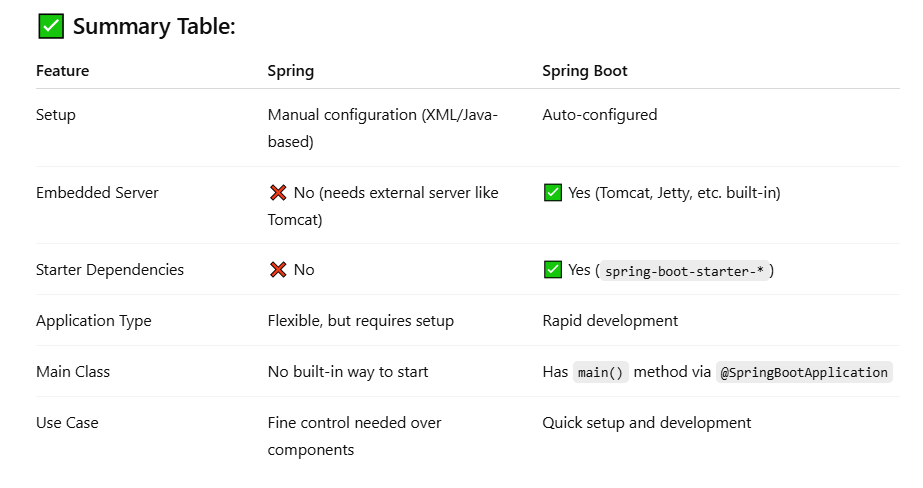
Kubernetes (K8s) Best for managing hundreds of microservices!

🔥 One-line Summary:

"In Microservices, each service is independently built, containerized (optional), and deployed separately using automation tools like Docker, Kubernetes,

or cloud services."

18)What is difference between spring and Spring boot ?



3. Spring Security JWT Authentication Workflow

🔸 Step 1: User Sends Login Request

The client sends a POST request with username and password to the /authenticate endpoint.

🔸 Step 2: AuthenticationManager Authenticates

Spring Security uses the AuthenticationManager to verify the credentials.

If credentials are valid, the user is authenticated.

🔸 Step 3: Generate JWT Token

After successful authentication, a JWT token is created.

This token includes the username and an expiration time.

The token is signed using a secret key to prevent tampering.

🔸 Step 4: Token Returned to Client

The JWT token is returned to the client (usually in the response body).

The client stores the token (typically in localStorage or sessionStorage).

4. Accessing Protected APIs

🔸 Step 5: Send Token with Requests

For subsequent requests to secured endpoints, the client includes the JWT in the Authorization header:

Authorization: Bearer <token>

🔸 Step 6: JWT Filter Intercepts the Request

A custom filter (JwtRequestFilter) runs before Spring Security filters.

It extracts the token from the header and validates it using the secret key.

🔸 Step 7: Validate and Set Authentication

If the token is valid, the filter sets the authentication in the SecurityContext.

Spring Security now considers the request as authenticated.

🔸 Step 8: Secure API Access Granted

The request proceeds to the controller, and the user can access the protected resources.

5. Key Components in the Architecture

Component Role

UserDetailsService Loads user-specific data during authentication.

AuthenticationManager Performs the actual authentication logic.

JWT Utility Class Generates and validates JWT tokens.

JWT Filter Intercepts each request, validates token, sets authentication.

Security Configuration Configures Spring Security to allow or restrict access.

6. Benefits of JWT Authentication

Stateless: No need to store session data on the server.

Scalable: Easily works in distributed systems and microservices.

Secure: Tokens are signed and optionally encrypted.

Flexible: Tokens can include roles, permissions, and other claims.

Can we print on console without main method- yes using static block

Yes, you **can print on the console without a main method** in Java **using a static block**, but this only works in certain versions of Java.

A **static block** is executed **when the class is loaded**, even before any method (including main) is called

public class Demo {

static {

System.out.println("Hello from static block!");

System.exit(0); // Exit to prevent looking for main method

}

}

**⚠️ Important Notes:**

* This works **only in Java versions before Java 7**.
* From **Java 7 onward**, the JVM strictly looks for a main method when running a class. If it's missing, you’ll get an error like:

Error: Main method not found in class Demo

21) Can we have multiple static block and what is the use of it

**Yes**, multiple static blocks are allowed.

**🔹 Key Points (Short):**

* **Executed at class loading time** (before main()).
* **Run in the order they appear** in the class.
* Used for **class-level initialization** (e.g., loading configs, initializing static variables).

**❌ Can we have multiple finally blocks in Java?**

**No**, Java does **not allow multiple finally blocks** for a single try block.

* You can have **multiple catch blocks**, but **only one finally block** is allowed per try.
* finally is used to execute **cleanup code** (like closing files or database connections), and it **always runs** whether an exception occurs or not.

**✅ Can we include a class as a key in a HashMap in Java?**

**Yes**, you can use a **Class object** as a key in a HashMap.

**🔹 Explanation:**

In Java, every class has a Class object (an instance of java.lang.Class). Since Class overrides equals() and hashCode(), it can be safely used as a key in a HashMap.

public class Example {

public static void main(String[] args) {

HashMap<Class<?>, String> map = new HashMap<>();

map.put(String.class, "This is a String");

map.put(Integer.class, "This is an Integer");

System.out.println(map.get(String.class)); // Output: This is a String

System.out.println(map.get(Integer.class)); // Output: This is an Integer

}

}

**🔹 Use Cases:**

* **Type-based mapping** (e.g., frameworks like Spring or Jackson use this pattern).
* **Service locator patterns** based on class types.

**✅ Conclusion:**

Yes, Class<?> can be used as a key in a HashMap because it is a valid Java object and properly implements equals() and hashCode().

**🔍 What is Reflection in Java?**

**Reflection** in Java is a powerful feature that allows you to **inspect and manipulate classes, methods, fields, and constructors** at runtime — even if you don’t know their names at compile time.

**🔹 Defined In:**

The **java.lang.reflect** package.

**🔹 What You Can Do With Reflection:**

1. **Inspect class structure** (fields, methods, constructors).
2. **Access private fields/methods**.
3. **Create objects dynamically**.
4. **Invoke methods at runtime**.
5. **Modify field values** at runtime.